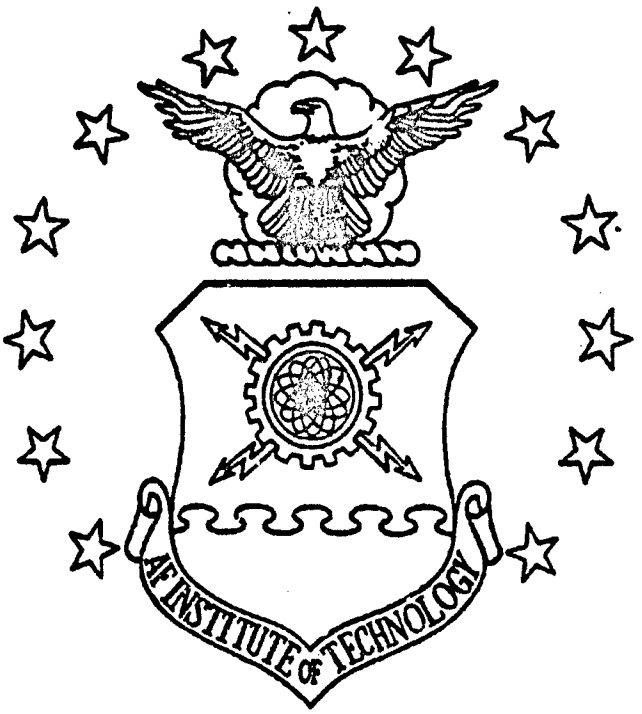


EASY_SIM: A VISUAL SIMULATION SYSTEM SOFTWARE ARCHITECTURE WITH AN ADA 9X APPLICATION FRAMEWORK

THESIS

Jordan Russell Kayloe
Captain, USAF

AFIT/GCS/ENG/94D-11

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/94D-11

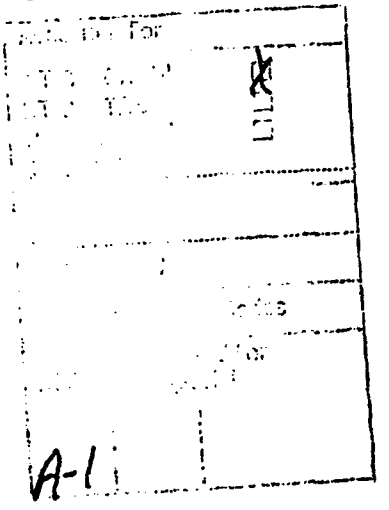
EASY_SIM: A VISUAL SIMULATION SYSTEM SOFTWARE ARCHITECTURE WITH AN ADA 9X APPLICATION FRAMEWORK

THESIS

Jordan Russell Kayloe
Captain, USAF

AFIT/GCS/ENG/94D-11

The views expressed in this thesis are those of the author and
do not reflect the official policy or position of the
Department of Defense or the United States Government.



AFIT/GCS/ENG/94D-11

EASY_SIM: A VISUAL SIMULATION SYSTEM SOFTWARE ARCHITECTURE WITH AN ADA 9X APPLICATION FRAMEWORK

THESIS

Presented to the Faculty of the School of Engineering of the

Air Force Institute of Technology
Air Univeristy

In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Science
with
Emphasis in Software Engineering

Jordan Russell Kayloe
Captain, USAF

December, 1994

Acknowledgments

The research that went into this thesis could not have been possible without the contributions of the numerous individuals enumerated below.

First and foremost I would like to thank my thesis advisor, Lt Col Patricia Lawlis. Her ideas and guidance were pivotal in the completion of this effort. I am extremely grateful to the time she has given me, and wish her luck as she retires from the Air Force and commences her new career.

I would also like to thank Dr. Thomas Hartrum and Capt Keith Shomper for being members of my thesis committee. Dr. Hartrum also gave me invaluable experience under fire with Ada 9X as the language expert, compiler tester, and all around guinea pig for his winter and spring Software Systems Generation classes.

My fellow classmates have provided moral support while I was toiling on this thesis. Capt Bob Caley, Capt Cathy Lin, Capt Steve Lindsay, and Capt John Vanderburgh also helped keep my head above water.

Four employees of Silicon Graphics, Inc. provided the tools for me to use Ada 9X on their machines, and they were gracious to host me as their guest over the summer. Tom Quiggle provided compiler solutions, Wes Embry and John Templeton supplied quite an example program, and Dave McAllister facilitated the whole effort.

The members of the GNAT Development Team at New York University are building a great compiler, and they have answered my silly questions about it tirelessly. So thank you to Bernard Banner, Dr. Cyrille Comar, Dr. Robert Dewar, Brett Porter, and Dr. Edmond Schonberg.

Jordan R. Kayloe

Table of Contents

List of Figures

List of Tables

Abstract

Software architectures increase productivity when used as the basis for developing applications in a problem domain. This thesis describes the creation of Easy_Sim, an object-oriented software architecture for visual simulation systems, and its corresponding implementation as an application framework in Ada 9X. The research built upon ObjectSim, an existing object-oriented simulation architecture implemented as a C++ application framework. Both ObjectSim and Easy_Sim operate on Silicon Graphics platforms and use the IRIS Performer graphics programming library. Easy_Sim is implemented using version 1.83 of the GNAT compiler.

The investigation for this thesis involved honing ObjectSim's design, implementing the improved result in both C++ and Ada 9X, and developing applications to compare the two versions. The study achieved two main objectives: producing Easy_Sim as an improved visual simulation system architecture by building on ObjectSim's experience, and producing a visual simulation system application from Easy_Sim in Ada 9X that performs at a level comparable to the same application built in C++.

EASY_SIM: A VISUAL SIMULATION SYSTEM SOFTWARE ARCHITECTURE WITH AN ADA 9X APPLICATION FRAMEWORK

I. Introduction

1.1 Background

A *visual simulation* is a system in which an operator is placed in a computer generated environment and presented with graphical stimuli mimicking actual objects and events. These systems have numerous commercial, military, and recreational applications because they provide safe, inexpensive, and efficient means of training personnel, serve as a logical test bed for innovative ideas, and have tremendous entertainment value. One of the most familiar visual simulation systems is the flight simulator, in which pilots step into an artificial cockpit and practice flying techniques without leaving the ground. The simulator allows the pilots to gain invaluable practice and train for adverse conditions without risking their safety, putting stress on real airplanes, or wasting valuable resources.

While there is no doubt that visual simulation systems are beneficial, their graphics-intensive software nature has presented numerous challenges throughout the years for system developers. The constantly increasing complexity of the simulated systems has led to a corresponding increase in the

software complexity of the simulators. In order to manage this complexity more easily, the building of reusable base designs, or *software architectures*, has begun in earnest by the makers of visual simulation systems. This trend is not unique to simulators; software developers in other problem areas, or *domains*, are also designing architectures to create solid foundations for their systems.

Software architectures can lead to many improvements in the development of software systems within a domain. Because an architecture promotes reuse at the design level, systems developers do not have to devote effort to analyzing and designing basic structures every time a new application of the system is needed. The existing groundwork allows developers to concentrate more of their work on problem specific areas, and cuts down on overall production time and cost [Law94,3]. The software architecture also makes the resultant system more maintainable and supportable, because changes can be incorporated more easily. By finding the place in the base design where a change will occur, its effects throughout the entire system can be determined and the impact of the change can be minimized. This ability to make changes more easily also leads to the practice of *rapid prototyping*, in which stepwise refinements are made to the implementation until the end result reaches production quality. Rapid prototyping allows system users to become more involved in the development process, giving them more control over the final outcome.

1.2 Problem History

The research for this thesis builds upon recent developments in creating software architectures for the visual simulation system domain. It primarily adds

to recent work accomplished in the Graphics Laboratory (Lab) of the Air Force Institute of Technology (AFIT). This graduate research involves *distributed* simulators, which allow a more realistic training situation by incorporating physically separated actors into a battlefield situation. The interaction between the various simulators is accomplished by means of network connections and established communication protocols. As part of this research, the AFIT Graphics Lab has recently produced a virtual cockpit for the F-15E fighter [Eri93, Ger93, Dia94], a space modeler [Kun93, Van94], a commander's situational battle bridge [Sol93, Wil93, Kes94, Roh94], and an air combat debriefing tool for the Air Force's Red Flag exercise [Gard93, For94]. Table 1 gives the details of the research projects completed in 1993.

Because these projects and their predecessors are visual simulation systems, they perform many of the same tasks. Historically, these common tasks were re-accomplished for each new application in the Graphics Lab. Under an initiative started in 1992, this unfortunate and inefficient problem was addressed by creating a software architecture dubbed *ObjectSim*. This design provided the common structure for each of the applications and allowed the different developers to concentrate more heavily on the unique aspects of their particular applications. ObjectSim also served as the focus of Mark Snyder's graduate thesis [Law94, Sny93].

The implementation of ObjectSim exists at a high level of generality, and can be used to create diverse simulation systems, as evidenced by the four applications. ObjectSim is also a completely object-oriented technology. In order to produce an application, developers must create their own versions of the structure by inheriting and deriving from the basic design. Because the ObjectSim implementation is abstract and can only serve as the basis for sys-

Table 1. AFIT Graphics Lab Research Projects for 1993

Project	Description	Research Goals
Virtual Cockpit [Erich93] [Ger93]	Immersive flight simulator for an F-15E	- Research inexpensive alternative to domed simulator - Build man-in-the-loop DIS platform - Study modeling of advanced weapons system
Synthetic Battle Bridge [Soltz93] [Wil93]	Immersive/Console based commander's eye view of battlefield	- Research immersive interface techniques for commander - Study expert computer situational analysis - Study situational representation techniques for battlefield - Study user interface techniques for effective user view control
Satellite Modeler [Kunz93]	Immersive/console simulation for analysis of satellites	- Represent single orbits or constellations - Study immersive interface into satellite simulation - Interface satellite data onto DIS simulations
Red Flag Display Tool [Gard93]	Console based debriefing/display tool for Air Force exercises	- Study user interface for debriefing system - Interface live or recorded exercise data onto DIS

tem applications, it is called an *application framework* [Sny93,12]. The reuse of ObjectSim's implemented code resulted in a vast increase in the production rate of visual simulation systems in the AFIT Graphics Lab [Sny93,78-82].

Unfortunately, ObjectSim does also have some drawbacks. Most notable is its heavy reliance upon underlying software libraries and hardware produced by Silicon Graphics, Inc. (SGI). The realistic speeds, or *frame rates*, with which ObjectSim applications are able to draw, or *render*, visual simulations can be attributed almost entirely to its use of SGI's IRIS Performer (Performer) graphics application development environment [McL92, Har94]. ObjectSim is

therefore not system independent, or *portable*. Another indirect consequence of ObjectSim's dependence on its SGI environment is its implementation in C+-. This choice was necessary to allow object-oriented extension and to achieve compatibility with Performer, whose interface is written in C. Many of the visual simulation system application developers have expressed that they could have been more productive had they not had to fight the cryptic intricacies of C++ [Law94,7].

1.3 Research Motivation

Ada is preferable to C+- as an implementation language for many life cycle software engineering reasons. These factors all cause Ada to be more expensive in terms of cost and time in the initial phase of a software project, but long run benefits justify these costs over the lifetime of the system. Ada code is inherently more readable than C-+ code, due to the more verbose and explicit nature of Ada syntax. More readable code is necessarily more understandable, and in an arena in which application developers build from a common framework, it is vital that they can comprehend that framework. More understandable code also contributes to Ada's higher modifiability, as the locations of changes are more readily apparent, and the effects of the changes throughout the entire program are more easily determined. In contrast, the cryptic notation of the C language which underlies all C++ code is notorious for producing unexpected results [Feu82].

Reliability is also much harder to achieve in C++ than Ada. C++ cannot dodge its C underpinnings, its weaker data typing facilities, or its currently non-standard implementation of exception handling, all factors that contribute to its lesser reliability. Even main proponents of C++ acknowledge its

disadvantages in this area. P. J. Plauger, the author of the ANSI C standard, has stated, "Beyond 50,000 lines of C, you'd better take a hard look at converting to C++. Beyond 100,000 lines, you should probably be coding in Ada" [Pla89, Pla94].

Mark Snyder was well aware of the benefits of Ada when he implemented ObjectSim, but he also had good reasons to choose C++. First, C++ was compatible with the underlying C tools of the SGI environment. Given that six other concurrent thesis efforts in the AFIT Graphics Lab relied on his work, Snyder did not have the time to finagle compatibility with Ada. Second, Snyder needed a language that supported the object-oriented extensibility he envisioned for ObjectSim. While Ada 9X supports this feature, it was not yet available. Snyder could only consider using Ada 83, which has no facility for extensibility. His choice for ObjectSim was therefore obvious.

Ada 9X has now become available through the development of preliminary compilers for the language. Although they are not yet complete, they do provide support for object-oriented extensibility and many other features that can enhance ObjectSim. Five more thesis efforts in the AFIT Graphics Lab have evolved the four applications, but they use Snyder's C++ version of the ObjectSim framework. The research on *Easy_Sim*, the Ada 9X version of the framework, therefore has been free to experiment with different versions of the implementation of the Easy_Sim architecture.

1.4 Scope of Research

This research has two primary goals: to develop a substantially improved Easy_Sim architecture using the knowledge gained from work accomplished with ObjectSim, and to demonstrate that a visual simulation system application

framework can be implemented using Ada 9X and provide capabilities equal to or better than a similarly designed C++ framework. The end result of the effort is therefore to demonstrate concept feasibility by producing an application using the Easy_Sim framework, and showing that the application's performance has not suffered. A C-+ implementation of the Easy_Sim architecture is maintained with the same functionality as the Ada 9X version. This version serves as a control so that a fair comparison is still possible if ObjectSim and Easy_Sim diverge substantially. Three questions were investigated in this research.

The first question investigated the design of ObjectSim. Mark Snyder's domain analysis consisted of examining components of each of the four applications under development during his tenure in the AFIT Graphics Lab, and determining which of these components might be useful to other applications. Even with this localized approach, Snyder's time constraints did not allow ObjectSim to encompass all of the functionality originally envisioned. Most notably absent is the handling of network interactions between the distributed simulators [Sny93,96-99].

The second question to investigate in producing a working application was the implementation of the software architecture design in Ada 9X. This question scrutinized the method for migration from C++ to Ada 9X structures. This question also involved ensuring that the proper tools were available to assist in the code production, including an Ada 9X compiler and bindings to the underlying SGI environment.

Once the Easy_Sim framework was built, the final question investigated the building of an application using Ada 9X. Mark Snyder describes example applications in the ObjectSim Application Developer's Manual [Sny93,AppA].

Implementing these sample programs in Ada 9X and analyzing their performance demonstrated the ability of the Easy_Sim framework.

Although it would be beneficial to divorce Easy_Sim from its reliance on the SGI environment, there are reasons this research did not entertain the idea. A high level comparison is a primary focus of this study, and it is therefore important that the frameworks be implemented similarly. The SGI environment is utilized because of its availability in the AFIT Graphics Lab and its proven capability for efficiently rendering graphics applications.

1.5 Methodology Overview

Each of the three questions outlined in the previous section corresponds to a set of actions that influenced the development of the Easy_Sim architecture and framework, as well as example applications.

The first question, investigating ObjectSim's design, corresponded to the domain analysis of the problem. The original analysis of ObjectSim was scrutinized for inconsistencies and deficiencies, and any changes deemed necessary or beneficial were incorporated into the design of Easy_Sim.

The second investigative question implemented the results of the domain analysis in Ada 9X. Different strategies for this process were considered, with an emphasis on reusing as much existing code as possible. The GNAT compiler was used to test these strategies. The first hurdle in the implementation process was the creation of Ada bindings to the Performer library. Luckily, visual simulation system developers at Silicon Graphics also have a keen interest in Ada 9X, and they created a preliminary set of bindings, as well as a port of the GNAT compiler. They generously agreed to contribute these products to this research effort.

The final investigative question compared the ObjectSim and Easy_Sim frameworks by demonstrating similar applications using each version. These demonstrations were initially accomplished by building Ada 9X versions of the ObjectSim example applications [Sny93.AppA]. Because the ObjectSim and Easy_Sim implementations differ, comparing them is unfair, and the C++ version of the Easy_Sim framework was used as a reference point.

In the discussion above, it mistakenly appears that the research occurred in three large chunks. In reality, investigation of the questions was performed repetitively in a rapid prototyping fashion, incorporating more of the solution's functionality with each iteration. The use of rapid prototyping allowed development of a working application earlier in the process. Each iteration corresponded to the addition of another feature in the architecture, with the existing ObjectSim example applications serving as product baselines.

1.6 Research Environment

This research effort used the equipment and tools located in the AFIT Graphics Lab. The SGI environment consists of various proprietary hardware and software systems. The machines include interconnected Indigo, Indigo^2, VGXT, and Onyx/Reality Engine^2 computer systems, with most of the work and all of the results collection accomplished on the four-processor Onyx known as Leonardo. The machines use version 5.2 of SGI's IRIX incarnation of the UNIX operating system. Version 1.2 of the IRIS Performer library [Har94] and version 5.2 of SGI's Graphics Library [McL91] enabled the graphics processing to occur at realistic rates. Version 3.2.1 of SGI's C++ preprocessing compiler was responsible for compiling the C++ code.

The Ada 9X code compilation occurred courtesy of GNAT, the Ada addition to the Free Software Foundation's gcc compiler family. A team at New York University (NYU) produces this shareware compiler with the support of the Ada Joint Program Office. Although the GNAT compiler is not complete, it has continually evolved and matured throughout the endurance of this thesis. The results presented were compiled with version 1.83.

Outside of the Graphics Lab, the contributions of various individuals was instrumental in the progression of this thesis. SGI's Ada team provided solutions to binding problems and ported the GNAT compiler to the SGI systems. The GNAT development team at NYU graciously responded to inquiries about the compiler and its maturity.

1.7 Document Overview

This research investigated developing a new version of a visual simulation system software architecture and implementing that architecture as an application framework in the Ada 9X programming language. The history, rationale, focus, and methods for this effort have been outlined throughout this chapter. The remaining chapters describe the research completely.

Chapter II details background topics pertinent to this effort. It covers object-oriented methodologies and programming languages, software architecture and application framework theory, industrial simulation architectures, the basics of the SGI environment, and the ObjectSim architecture. Chapter III illustrates the changes made to the ObjectSim domain analysis and design to produce Easy_Sim. Chapter IV presents the techniques used in the migration from the C++ implementation of ObjectSim to the Ada 9x implementation of Easy_Sim. Chapter V describes the development of visual simulation system

applications built from Easy_Sim. Chapter VI analyzes the results of the comparison of the different versions of the architecture. Finally, Chapter VII summarizes the research accomplishments and suggests areas for improvements and future study.

II Overview of Current Research

Before discussion of the details of this thesis occurs, this chapter presents various prerequisite topics. The chapter is constructed so that the reader can easily skip any of the concepts with which he or she is already familiar.

The first section of this chapter gives an introduction to object-oriented software development by looking at different related topics. It first covers the overall concepts that define the term *object-oriented*, and then discusses the implementation of the object-oriented methodology in both the C++ and Ada 9X programming languages. The final portions of the first section cover the Rumbaugh technique for describing object-oriented analyses and designs, and the ROMAN-9X method for developing Ada 9X code from Rumbaugh diagrams.

The second section of this chapter defines software architectures, and the third section analyzes these architectures as they are used within the simulation industry today. The fourth section describes the Silicon Graphics Performer library, and the final section introduces ObjectSim, the architecture upon which Easy_Sim is based.

2.1 Object-Oriented Concepts

Object-oriented methods, including analysis and design techniques as well as programming languages, have certain characteristics that give them a clear advantage in the software development process. Perhaps the most obvious of these is their reliance on an *objective* problem view instead of the more traditional functional view. People are more easily able to identify the concrete players, or *objects*, involved in solving a problem than they are able to imagine the abstract processes needed to achieve a result. Object-oriented pro-

gramming paradigms capitalize on this more intuitive way of attacking a problem, and have been more readily received because of it. Software developers first determine which objects they need in a problem, and then figure the processes, or *operations* that go with each object.

While an object-oriented method may be more intuitive, it would be useless if it did not lend itself to the production of systems that adhere to the common goals of software engineering [Ros75]. Luckily, maintainable, understandable, reliable, and efficient code can be readily produced through object-oriented programming techniques. Technically, most experts agree that there are four major features that make a methodology object-oriented: abstraction, encapsulation, inheritance, and polymorphism [Boo94, Rum91, Str91]. Each of these features contributes in the effort to attain the goals of software engineering.

Abstraction is the separation of an object's specified functionality from its implementation. One programmer writes a set of routines with a well-defined interface, and another programmer easily integrates that code into her own work with simple calls. The caller's code is unaffected by changes in the implementation, and she can therefore be completely oblivious of the routine's implementation. This feature is preferable in a large project, in fact, because it is impossible for a single programmer to understand the entire system. Types that use abstraction have historically been termed *abstract data types (ADTs)*, and to effectively use this powerful concept, the programmer must learn to rely on *what* a piece of code does, instead of *how* exactly it works. Ironically then, abstraction allows software developers to accomplish more by understanding less. The reality of this situation makes abstraction the key to objects.

The second major feature of object-oriented languages, *encapsulation*, builds upon abstraction. Encapsulation is also often commonly referred to as *information hiding*. Where abstraction logically separates what happens with an object from how it occurs, encapsulation physically divorces the two ideas. A piece of code's implementation is no longer simply irrelevant to a client programmer, it is also inaccessible, grouped together and hidden within the code's innards. There are two basic scenarios where this is invaluable for a system's reliability. The first is to prevent hackers, who know the intricacies of the implementation, from intentionally trying to affect the outcome of a routine. Tricks or kludges that they think enhance the code may unfortunately introduce errors into the work of other client programmers. The second scenario occurs when a maintenance programmer is charged with enhancing a piece of code, but only partly understands its implementation. The programmer makes some minor changes in a module that seem to achieve the desired result. While he is not intentionally trying to change the overall implementation of the system, he may inadvertently end up doing so and once again affect the whole system. Encapsulation is therefore a feature that can increase reliability by grouping all related code together. It serves as a safety mechanism preventing both malicious and accidental breaches of an abstraction.

Inheritance is the third major feature of object-oriented languages. It follows from the real world model of objects, where one object can be very similar to another object, only with a few new additional details. An FM stereo radio, for instance, is an FM radio with left and right channels. Its basic functionality is the same--it produces sound. However, FM stereo radios may also have balance controls or a stereo indicator light, *attributes* not found in a

regular FM radio. Inheritance works just this way, taking the basic functionality from one type of object and extending it to a new type of object, adding new attributes to the new type if necessary.

Object-oriented terminology states that a particular object is an *instance* of a type of object, or *class*. A class consists of all the types *derived* from it, including itself. The derived classes are often called *subclasses* or *children* of the *base* or *parent* class, and together they form a *derivation tree*. FM stereo radios are therefore subclasses derived from FM radios, which could in turn be a child of a generic radio class. Often, as in this example, a derivation tree's root class is *abstract*, merely factoring out functionality common in all of its subclasses, and instances of the abstract base class cannot exist. In this example, the abstract radio class has power switches, tuning controls, and volume controls, but creating an instance of the radio class does not make sense unless it also receives some band of frequencies, such as FM, AM, or short wave. An instance of the abstract radio class, therefore, must belong to one of its concrete subclasses.

Inheritance is very important in improving the maintainability of a system. It makes the task of extending a module's behavior independent of the module itself. In other words, the new functionality can be added in a new module without making any changes to the original. This concept is helpful for three main reasons. Whenever code is changed, the possibility of unintentionally introducing errors always exists, and reliability may therefore suffer. Second, if a piece of code does not change, it does not have to be tested again. When inheriting code, the programmer only has to test the extensions in the new module, saving time, effort, and cost. The final reason for avoiding working in the original module lies with compilation dependencies. If a mod-

ule is recompiled, some language systems require that all other dependent modules also be recompiled to incorporate any possible changes. This extra step is necessary regardless of whether or not the dependent module actually used the new capabilities. Obviously this task can slow down, or even retard, development, and is an annoying hindrance to a project's productivity. The object-oriented feature of inheritance is therefore a welcome enhancement for any software maintainer, saving her from numerous unnecessary annoyances [And93].

Just as the feature of encapsulation adds to abstraction, the final major feature, *polymorphism*, adds to inheritance. Returning to the earlier example, all radios share a basic operation--they produce sound. Analyzing the radio class, radio users are always given a method for turning a radio's power on and off to produce this sound, whether they have an AM, FM, or short wave radio. The switches to do this operation may differ depending on the subclass of radio, and the inner workings of the radio's power source may vary, but power switches are common throughout the entire class. This commonality is polymorphism, the inductive idea that although an operation is invoked similarly for different subclasses, the operation may behave differently for each class. Technically, the system makes a *dispatching* operation, determining how to perform the operation depending on the particular subclass upon which the operation is called. If the system cannot determine the subclass with which it is dealing until run-time, the dispatch is performed *dynamically*.

Polymorphism helps us achieve the goal of understandability in the same way function overloading does, because similar operations are given similar understandable names. If the main radio operation is "Power_On," this name is used throughout the derivation tree no matter how the operation is

performed. This convention has a positive effect on maintainability for modules that call a dispatching operation. If a new subclass is added to a derivation tree, absolutely no change will have to be made to the existing call. If the call is made with an object of the new type, it will still dispatch accordingly.

A general trend is very noticeable in the discussion of object-oriented features. Programmers continually mention using other programmers' objects. The objects can call each other easily due to well-defined abstractions and not worry about the encapsulated details. They can design by inheriting from and extending another programmer's design, allowing calls to be made to either design in the same manner with polymorphism. This idea of *reuse* is central to object-oriented programming and is its "most tangible advantage" [Ban92]. Once an object and its operations are defined, it can be shared among different developers in one project, or even across different projects altogether. There is no need to try to modify an object if it already works. If more capabilities are needed, simply inherit from the established to create the new.

Reuse is not a new idea, but object-oriented programming languages can realize its benefits without the recompilation necessary in more traditional languages. This feature cuts down tremendously on production time and translates directly into development and maintenance savings. Dr. Edmond Schonberg of New York University has said about object-oriented programming, "the gain is in the amount of code that one does not have to write" [Sch92]. This concept alone is perhaps the most compelling reason to readily accept the object-oriented paradigm. Combined with the more intuitive methodology for breaking down a problem and the improvements brought to the software engineering goals of maintainability, understandability, relia-

bility, and efficiency, it is clear why object-oriented methods are revolutionizing the software industry.

Discussion now turns to the two programming languages that are used in this thesis effort to realize the object-oriented paradigm, C++ and Ada 9X.

2.1.1 Object-Oriented Programming in C++

The C++ language serves as an extension to the C language, and both are currently used widely throughout industry. C is notorious for allowing programmers to produce unstructured code, and C by itself lends little support for any of the design principles associated with object-oriented programming. This section addresses the features of C++ that support the object-oriented paradigm, giving structure to the C language family [Str91, Poh93].

The *class* typing construct in C++ directly maps to the notions of abstraction and class in object-oriented terms. A C++ class defines a type that can be used by client programmers. Its definition includes any attributes, or *members*, as well as any operations, or *member functions*, that might act on an instance of the class. The term member is used in both these cases to indicate that both the attributes and operations are declared within the scope of the class [Ker78,120]. Instances of the class are achieved through variable declarations in a client program, and creation and deletion of these objects can be controlled by the class designer through the use of automatic *constructors* and *destructors*. The separation of specification and implementation that constitute abstraction is accomplished in C++ through the use of *header* and *source* files, which respectively contain the two views of an object. The header file in Figure 1 defines the interface for an FM radio.

The source file for the FM_Radio class, shown in Figure 2, defines how the member functions achieve their effects. These implementations can only be accessed by clients through the interface defined in the header file. The "::" notation gains access into the scope of the class preceding the symbol, and is necessary because many classes can have member functions with the same name [Str91,145]. It is also vital because C++ does not define rules for locating the definitions of the member functions in any specific source file. Note how no part of the language determines the beginning or end of the header file--it is, by convention, a simple list of the definitions of the member functions but it could easily contain other entities.

```
// file fm_radio.h
class FM_Radio
{
public:  // Grants client programmers access to what follows

   FM_Radio ();   // Constructor has same name as class
   ~FM_Radio ();  // Destructor's name is similar, but with tilde

   // Member functions:
   // Syntax-- return-type name (parameter-type parameter-name);

   void Power_On  ();   // Void functions return no values
   void Power_Off ();   // Empty parameter lists must be explicit

   void Volume_Up   ();
   void Volume_Down ();

   void Tuning_Up   ();
   void Tuning_Down ();

   float Station ();


   // Regular members, storing state of class:
   int   power;            // 0 is off
   int   current_volume;
   float current_station;

};     // Semi-colon completes class declaration
```

Figure 1. FM_Radio Class Header File

```
//  file fm_radio.cc

#include "fm_radio.h"

FM_Radio::FM_Radio ()
{
   power          = 0;
   current_volume = 0;
   current_station = 97.7;
}                                  //  No semi-colon here

//  Other member functions omitted

FM_Radio::Power_On ()
{
   power = 1;
}

//  and so on...
```

Figure 2. FM_Radio Class Source File

Instances of a C++ class each get their own copies of all its members. When a member function is called in a client program, the instance name is part of the call, and the member data upon which the function operates is passed implicitly. Figure 3 shows a main C++ function that uses the FM_Radio:

Just as the FM_Radio header file above declares a part of the file to be public, it can declare a part to be *private*. This feature brings encapsulation to

```
//  file main.cc

#include "fm_radio.h"      //  To access class header file

int main ()
{
   FM_Radio   *My_Radio;  // Declares pointer to instance of FM_Radio

   My_Radio = new FM_Radio (); // Allocates space & calls constructor

   My_Radio->Power_On ();        //  Turns on My_Radio instance
   My_Radio->Volume_Up ();       //  Turns up My_Radio instance
}
```

Figure 3. Main Using FM_Radio Class

C++, hiding whatever is declared in the private part from clients of the class, and only allowing access to member functions. In fact, private is the default, and all members will be encapsulated unless explicitly listed otherwise. Normally, a class encapsulates its regular members in the private part, while keeping its member functions available in the public part. The definition of member functions in a separate source file also contributes to the encapsulation of C++.

Any C++ class can form the root of a derivation tree, and this feature brings inheritance into the language. Another class can simply declare that it is a child of FM_Radio, and it gets a local copy of all the members of the parent class. The derived class does not gain any special privileges, however, and any members of the base class declared private are inaccessible to the child. If this effect is not desired, the parent can have a part similar to its public and private parts, called *protected*, and any members declared in this part are visible throughout its descendant subclasses. A child class can add member data elements to those inherited from its parent by simply declaring more of its own, in whichever of its own parts it prefers.

The subclass cannot, however, customize its parent's member functions unless the parent explicitly grants permission for this polymorphism to occur. A class can declare any of its member functions to be *virtual*, allowing that function to be overridden or redefined by its subclasses. Additionally, a class can make itself abstract by indicating that one or more of its virtual functions is *pure*. Pure virtual functions cannot have definitions in the class in which they are members, and no instances of a class with pure virtual functions can be created. Any descendant of an abstract class must override the abstract

```
// file radio.h

class Radio
{
public:

   Radio ();       //  Inlined null function acts as destructor
   ~Radio () {};   //  Simple functions can likewise be inlined

   virtual void Power_On  ();
   virtual void Power_Off ();

   virtual void Volume_Up   ();
   virtual void Volume_Down ();

   virtual void Tuning_Up   () = 0;  // "= 0" indicates pure function.
   virtual void Tuning_Down () = 0;  // Frequency band is unknown.

   virtual int   Station () = 0;
   virtual float Station () = 0;

protected:

   int   power;            //  0 is off
   int   current_volume;

};
```

Figure 4. Abstract Radio Class Header

functions, unless it too is intended to be abstract. Figure 4 shows the abstract Radio class, and Figure 5 demonstrates how the FM_Radio is derived from it.

The main program can now declare instances of any class derived from Radio, and expect that if a Tuning or Station member function is called, the appropriate routine will be executed depending on the subclass of the instance. Figure 6 shows a function which demonstrates this polymorphism.

In this example, the parameter passed to Turn_Off_Radio is a pointer to any subclass of the abstract Radio class. When making the call to Power_Off, the run-time system will determine the actual subclass of the parameter and dispatch the call to the Power_Off function for that subclass, be it an FM_Radio, an AM_Radio, a Short_Wave_Radio, or a Banana_Radio, whatever that may be.

This section has provided a brief overview of the C++ language's support

```
// file fm_radio.h

#include "radio.h"

class FM_Radio : public Radio   // Derived from Radio
{
public:

   FM_Radio ();   // Override constructor

   // Destructor, Power, and Volume functions are inherited as is

   virtual void Tuning_Up   (); // Redefine pure functions, and make
   virtual void Tuning_Down (); //   virtual to allow overriding

   virtual int   Station ();
   virtual float Station ();

protected:

   float current_station;          // Add new frequency band

};
```

Figure 5. FM_Radio Subclass Header

for the object-oriented programming paradigm. More in depth treatment of the subject is available from many sources [Str91, Poh93]. Discussion now continues with a similar implementation of the radio hierarchy in Ada 9X.

2.1.2 Object-Oriented Programming in Ada 9X

The Ada programming language became an ANSI standard in 1983, with the intent that it would be updated periodically as programming methodologies evolved. Ada 9X is the first of these updates, supplementing the original lan-

```
void Turn_Off_Radio (Radio *My_Radio)
{
   My_Radio->Power_Off ();   // Dynamic dispatching call
}
```

Figure 6. Dispatching Radio Function

guage with many new features deemed necessary by its users. Ada 9X was approved by the International Standards Organization in November 1994, and will be officially dated according to the printing date of the new Ada Language Reference Manual. Upward compatibility has been a primary goal of the revision process, and because Ada 9X fully embraces Ada 83, this objective has been successfully achieved.

Unlike the C basis of C++, the Ada 83 basis for Ada 9X already provides a sound basis for object-oriented principles, fully supporting both abstraction and encapsulation. It also contains a limited form of inheritance, but does not readily allow polymorphism. This section looks at the Ada features that address the four object-oriented principles. A trait common solely to one of the Ada versions is clearly indicated, while mention of an "Ada" feature indicates a feature common in both versions.

Because the Ada language was originally intended to be used on large software development projects, its designers decided to provide extensive capabilities for abstract data types. Ada realizes this well-proven programming concept by an idiom using both its *private type* and *package* features. An Ada package can serve as a container for many programming entities, but it is also a tool for abstraction. Just like the C++ class, an Ada package physically separates its interface from its implementation, and these parts are respectively called the *specification* and *body* of the package. Ada mixes its rich typing facilities with packaging in the form of private types, which split a package's specification into public and private parts. The private type and any operations that manipulate its values are declared in the public part, so that client programmers can access them. The components that the private type comprises are then defined in the private part of the package specification, and

are encapsulated so that the client programmer cannot access them. The body of the package, which defines the implementation of the operations that manipulate values of the private type, has full access to the specification's private part, but is also hidden from any client programmers. Ada packages that export private types are sometimes called *class packages*, since the abstract data type corresponds to a class in the object-oriented paradigm [Cer93]. In fact, experts often call Ada 83 an *object-based* language because of its abstract data typing facilities [Boo94, Taf92a].

Returning to the example of the FM radio, Ada syntax corresponding to the C++ code for the FM_Radio class appears in Figure 7.

The most notable semantic difference between the C++ code and the Ada code is the use of parameters in the operations of the FM_Radio class. Ada requires these parameters as a consequence of its support for concurrency. Because an Ada procedure or function must be able to execute flawlessly when many copies of it are running concurrently, each copy must get its own copy of the data upon which it is operating.

The result for the client programmer is not very dire, as the names used in the call simply appear in a different order. The main difference in the client is the reference that is made to the package where the type and its operations are defined. Ada uses this explicit reference to increase maintainability on large programs, where tracing a declaration can be cumbersome. This explicit referencing can be circumvented if the programmer so desires, but this practice is discouraged and is not shown in Figure 8.

```
-- file fm_radio.ads

-- Naming conventions for objects taken from ROMAN-9X [Cer93]
-- See appropriate section below for more information.
package FM_Radio is

   type Object is private;       --  Private type declaration

   --  Operation declarations
   --  Syntax-- procedure [function] name
   --              [parameter-name : parameter-modes parameter-type]
   --              [return return-type];
   procedure Power_On  (Instance : in out Object);
   procedure Power_Off (Instance : in out Object);

   procedure Volume_Up   (Instance : in out Object);
   procedure Volume_Down (Instance : in out Object);

   procedure Tuning_Up   (Instance : in out Object);
   procedure Tuning_Down (Instance : in out Object);

   function Station (Instance : in Object) return Float;

private

   --  Encapsulated types needed for full private type
   type    Switch    is (Off, On);
   subtype Frequency is Float   range 87.7..107.9;
   subtype Volume    is Natural range 0..10;

   --  Full private type definition
   type Object is
      record  -- default initial values provided for components
         Power          : Switch    := Off;
         Current_Station : Frequency := Frequency'First;
         Current_Volume  : Volume    := 0;
      end record;

end FM_Radio;


--  file fm_radio.adb
package body FM_Radio is  --  Body tells compiler what it is

   procedure Power_On (Instance : in out Object) is
   begin
      Instance.Power := On;
   end Power_On;

   --  And so on...

end FM_Radio;
```

Figure 7. FM_Radio Class Package

```
-- file listener.adb
with FM_Radio;  --  To access public package contents
procedure Listener is
   My_Radio : FM_Radio.Object;   --  Declares instance of FM_Radio
begin
   FM_Radio.Power_On  (My_Radio);
   FM_Radio.Volume_Up (My_Radio);
end Listener;
```

Figure 8. FM Radio Listener Procedure

Ada 83 permits inheritance of an object's attributes, inheritance of an object's operations, and extension of an object's operations through its *derived* types. A type declared in a package specification is automatically *derivable*, as are any subprograms in the public part of the package specification that take a parameter of the type in question. These subprograms are called the type's *primitive operations*. A client program unit can derive a new type from the original, and the child inherits the attributes and operations of the parent. The child may override any of the inherited subprograms as necessary, and any subprograms it declares that take a parameter of the derived type are further derivable in other program units.

Ada's derived types do not, however, allow extension of an object's attributes, a concept vital to object-oriented programming. Ada 9X introduces a new kind of record type to correspond to a class, called the *tagged type*, that allows new record components to be added to any type derived from it. Tagged types therefore provide full support for inheritance. The new components that correspond to the attributes of the subclass may be specified in the public part, but a design fully adhering to the idea of encapsulation declares the new attributes with a *private extension*, defined in the package's private part.

Tagged types may also be abstract, and may declare abstract operations that must be overridden in child classes. Finally, a tagged type may be *controlled*, providing a default constructor and destructor, if it is derived from a predefined abstract tagged type called Ada.Finalization.Controlled. Ada 9X's controlled types also allow value *adjusting*, so that assignment between different instances of a class can also be controlled by the programmer. If assignment is not desired between instances of a class, the class can be derived from Ada.Finalization.Limited_Controlled. These *limited controlled* types only inherit a constructor and destructor. The package specification in Figure 9 shows the Ada 9X version of the controlled, abstract Radio class.

Just as in the C++ implementation of classes, Ada 9X does not allow a child class default access to its parent's private part. Unlike C++, however, Ada 9X does not grant the parent class the ability to change this sometimes bothersome feature. In Ada 9X, the derived type takes control by exploiting the new feature of *hierarchical library units*. The new package declares itself to be part of another package, so that it is logically nested inside its owner, even though it is physically separated. Because Ada 9X's new type of package is used in combination with inheritance, it is often referred to as a *child library unit*, or simply a *child package*. In visibility terms, the child package does not need to access its parent package using an explicit *with* clause, because the compiler recognizes the child's intent to be part of its parent. The public part of the child is logically located at the end of the parent's public part, and the private part of the child is logically located at the end of the parent's private part, so that the child has access to the entire parent. The package specification for the FM radio child class package follows in Figure 10.

```
-- file radio.ads

with Ada.Finalization;  -- To provide controlled capabilities

package Radio is

   -- Allow visibility of attribute types:
   type    Switch is (Off, On);
   subtype Volume is Natural range 0..10;

   -- Make subclass of base for controlled types:
   type Object is abstract new Ada.Finalization.Controlled with
      private;

   -- Controlled operations:
   procedure Initialize  (Instance : in out Object);
   procedure Adjust      (Instance : in out Object);
   procedure Finalize    (Instance : in out Object);

   procedure Power_On    (Instance : in out Object);
   procedure Power_Off   (Instance : in out Object);

   procedure Volume_Up   (Instance : in out Object);
   procedure Volume_Down (Instance : in out Object);

   procedure Tuning_Up   (Instance : in out Object) is abstract;
   procedure Tuning_Down (Instance : in out Object) is abstract;

   function Station (Instance : Object) return Natural is abstract;
   function Station (Instance : Object) return Float   is abstract;

private

   type Object is abstract new Ada.Finalization.Controlled with
      record
         Power          : Switch := Off;
         Current_Volume : Volume := 0;
      end record;

end Radio;
```

Figure 9. Controlled, Abstract Radio Class Package Specification

Just as tagged types bring the object-oriented principle of inheritance to Ada 9X, they also bring polymorphism to the language. The name *tagged*, in fact, refers to the polymorphic qualities of tagged types, as the system maintains a *tag* to keep track of an instance's subclass, so that it can dispatch to the proper primitive operation. Because each tagged type forms the root of a class derivation tree, Ada 9X provides a new language type attribute for tagged types

```
--  file radio-fm.ads

package Radio.FM is

   subtype Frequency is Float range 87.7..107.9;

   type Object is new Radio.Object with private;

   procedure Initialize (Instance : in out Object);

   -- Other Controlled, Power, and Volume operations inherited

   -- Abstract operations must be overridden:
   procedure Tuning_Up   (Instance : in out Object);
   procedure Tuning_Down (Instance : in out Object);

   function Station (Instance : Object) return Natural;
   function Station (Instance : Object) return Float;

private

   type Object is new Radio.Object with record
      Current_Station : Frequency := Frequency'first;
      -- Power, Current_Volume inherited
   end record;

end Radio.FM;
```

Figure 10. FM Radio Subclass Child Package Specification

called *T'Class*. This attribute refers to any subclass in the hierarchy started at type T, and allows the declaration of *unconstrained* objects that can take the form of any type derived from T. These objects are accordingly called *classwide objects*, and can be declared wherever an object declaration is appropriate. Figure 11 shows a modification to the Listener procedure previously seen in Figure 8. Listener is now a *classwide operation* because it takes a classwide object as a parameter to exploit the dynamic dispatching available in Ada 9X.

Both of the procedure calls above perform dynamic dispatching. The procedure corresponding to the Instance's tag will be called by the system, and this procedure may not necessarily be the one defined in Radio. Classwide operations normally call the operations of the root type for understandability as

```
--  file Listener.adb

with Radio;                    --  To access root of hierarchy
procedure Listener (Instance : in out Radio.Object'Class) is

begin

   Radio.Power_On  (Instance);
   Radio.Volume_Up (Instance);

end Listener;
```

Figure 11. Classwide Radio Listener Procedure

shown here, but run-time polymorphism allows any primitive operation in the classwide object's hierarchy to be used [Taf92a].

While classwide objects are quite useful, in object-oriented programming it is often more practical to deal with pointers to objects. Ada 9X provides *classwide access types* to implement this functionality, and these types are normally included in a class package to provide additional capability. A classwide access type declaration appears:

```
type Reference is access all Object'Class
```

Assuming this line exists in the Radio class package, the Listener procedure can be changed to handle pointers, with an adjustment made in the actual parameter passed to the calls accounting for the pointer dereference. Figure 12 shows the new version of Listener. Seeing pointers used in this fashion, and realizing that the .all dereference is not aesthetically pleasing, it would seem that the next logical step would be to change the parameter types of the derivable operations to use classwide access types instead of tagged types. This move would be quite erroneous, however, because it is the tagged type parameter itself that makes the operations derivable and allows dispatching to occur. Changing the parameter type of the primitive operations to a classwide access

```
-- file listener.adb

with Radio;
procedure Listener (Instance : in out Radio.Reference) is
begin

   Radio.Power_On  (Instance.all);
   Radio.Volume_Up (Instance.all);

end Listener;
```

Figure 12. Classwide Radio Listener Procedure with Access Types

type is unnecessary, as the intent of the change is already provided with the simple tagged type. The somewhat ugly result of this rule is the necessity to keep the .all.

This section has provided a brief overview of the Ada 9X's support for the object-oriented programming paradigm. More in depth treatment of the subject is available from many sources [Bal93, Bar93, Bar94, Cer93, Coh93, Kam93, Kem94, Rat94]. Discussion now turns to one method for analyzing a problem in an object-oriented fashion, independent of programming language.

2.1.3 The Rumbaugh Object Modeling Technique

Dr. James Rumbaugh and his colleagues at the General Electric Research Center have devised an object-oriented approach to attacking the analysis and design phases of the system life cycle. This methodology is called the *Object Modeling Technique*, and results in a design that is independent of both programming languages and hardware platforms [Rum91]. Although the Object Modeling Technique covers many aspects of the analysis and design phases, this section centers on the *Rumbaugh diagrams* that are used to show the relationships among the objects in a system. Figure 13 shows a Rumbaugh diagram of the Radio system. In order to illustrate the features of the Object

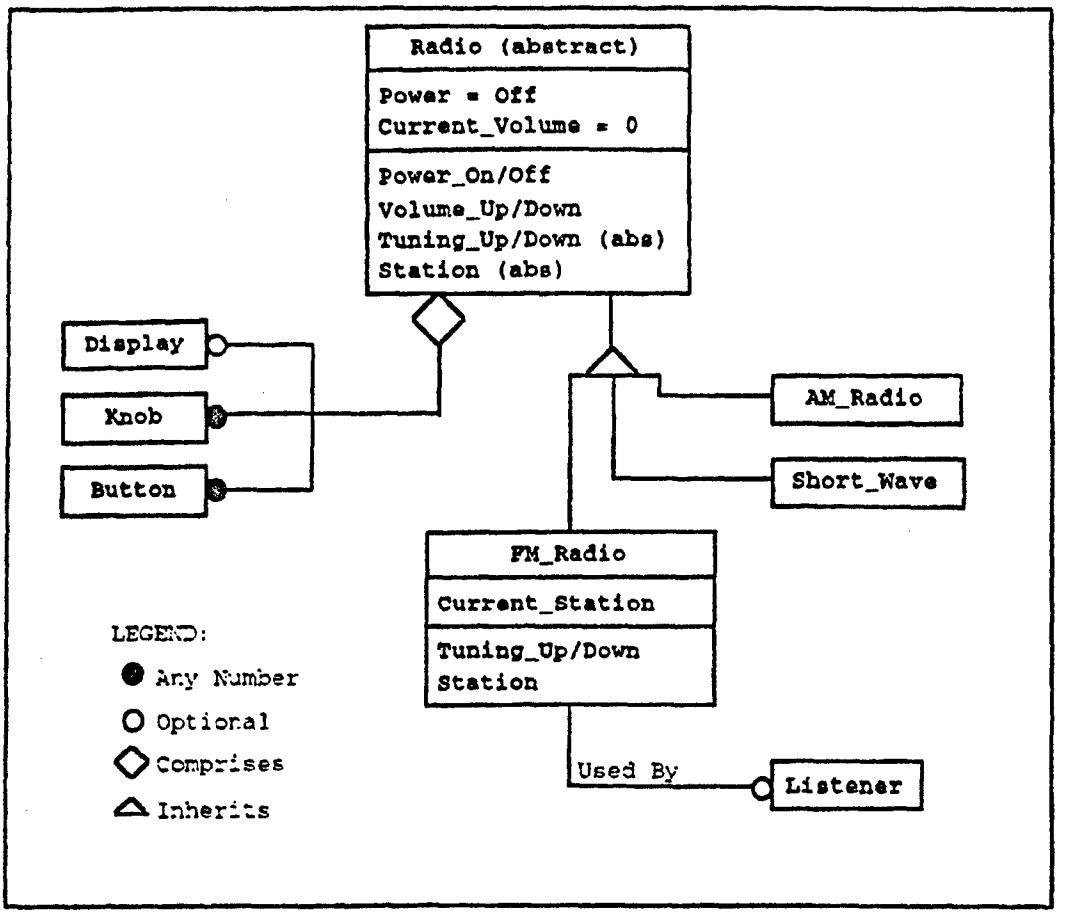


Figure 13. Rumbaugh Object Model Diagram for Radio Hierarchy

Modeling Technique that are necessary for this thesis, additional classes are shown that have not been previously discussed.

Each box shows a class within the hierarchy, with the optional three subdivisions respectively showing the name, the set of simple attributes, and the set of operations that belong to the class. A simple attribute may denote an initial or default value, and an operation may specify any parameters needed to perform its task. There is no specific way to show an abstract class, so this characteristic may be written explicitly. In the figure, the abstract Radio class

has two attributes, both of which have default values. The operations of the Radio class are shown in pairs, and the abstract operations are noted. In the FM_Radio class, only the new attributes and the new or overriding operations are indicated.

Lines connecting the classes represent the relationships that exist between them. Inheritance is shown with a triangle, and the diagram above therefore demonstrates that the AM_Radio, Short_Wave, and FM_Radio classes are all derived from the Radio class. *Aggregation* is a special relationship in object-oriented analysis that indicates that a class has attributes that are themselves classes. These attributes are separated from simple value-oriented attributes because more information is required to understand their values and operations. In the diagram above, the Radio class and all its descendants are now composed of Displays, Knobs, and Buttons, in addition to the simple attributes already mentioned.

While inheritance and aggregation are common, relationships between classes can exist that do not have special characteristics. A normal line shows that an instance of a class on one end must coexist with an instance of the other class, and these lines are normally labeled to describe the relationship. Different *multiplicity balls* dictate the number of instances that must be related in the real system. An outlined ball indicates an optional relationship, while a solid ball shows that any number of instances of the class can be related. In the Radio Object Model, the Listener class must use an FM_Radio, although the FM_Radio can stand on its own. The three classes that make the aggregate for the Radio also have multiplicity balls. They indicate that a Radio may or may not possess a Display, can have any number of Knobs, and can contain as many Buttons as it needs.

This section has introduced the basics of the Rumbaugh Object Modeling Technique. More details can be found in the text that Dr. Rumbaugh and his colleagues have published [Rum91]. Discussion now addresses a technique for developing Ada 9X code from Rumbaugh Object Models.

2.1.4 Representing Object Models in Ada 9X Notation

This section serves as a summary of a technique for *Representing Object Models in Ada 9X Notation (ROMAN-9X)* [Cer93]. It serves as the basis for converting Rumbaugh diagrams into Ada 9X package specifications. This design methodology has already been used throughout the discussion above to implement the radio hierarchy, but is explained more explicitly here. Some additions to the technique have been made, and they are included in the discussion below.

The basis of ROMAN-9X is that each class is implemented in its own *class package*, a module which wholly and distinctly contains everything particular to that class. To aid program readability, the package name is chosen carefully to serve as the name of the class. The *class type* represents the class itself and is called Object. The class type is implemented as a tagged type to allow inheritance, and is usually controlled to allow for constructors and destructors. The Ada type mark for client programmers declaring instances of the class is therefore Class_.Name.Object. This name reads nicely, and avoids confusion by equating the class name with the Ada type itself. To provide flexibility in using the class, the class package provides a classwide access type named Reference. This type allows client programmers to easily and efficiently use the class as an attribute, and allows more flexibility in the exploitation of polymorphism.

Class operations are declared as procedures and functions in the public part of the class package specification. Default initialization is accomplished either through explicit initialization in the record component declaration or by Initialize if a controlled type is used. In cases where parameters must be passed to complete initialization, perhaps because of dynamically determined data values, a procedure called Configure takes in the necessary values. By convention, the class object is the first parameter to any operation, and the formal parameter is named Instance. This standard idiom aids readability and provides a similar way for all operations to refer to the instance of the class object upon which they are operating.

Class attributes are encapsulated in the tagged record in the private part of the package specification. The class declares Get and Set operations to access the class attributes when necessary, and the attributes are never accessed globally. The Get operations are implemented as functions when possible, and are given the name of the attribute they return. If the Get operations returns more than one value and must therefore be a procedure, it is named Get_(Attribute). The Set operations are always procedures, and are named Set_(Attribute). Both the Get and Set operations are inlined whenever possible. If the class is an aggregate, and it has attributes that are themselves classes, these attributes are stored as References, and the class package that defines the attribute must be accessed via a *with* clause. Using a classwide access to the attribute allows the attribute to be passed easily and efficiently and avoids the potential dispatching conflict of having two distinct tagged types in an operation's parameter list. It also allows the attribute to take on the value of any subclasses that may be derived from the class originally envisioned.

A derivation tree is rooted at a class package using the hierarchical library units of Ada 9X. The root of the tree serves as the base class package, with each descendant implemented in a child package. A hierarchy with an abstract root is implemented using the corresponding new Ada 9X syntax for abstract types. Within the hierarchical packages, the package name indicates a class' position, with each dot indicating its depth within the derivation tree.

Having gone from the basics of the object-oriented methodology to a technique for creating object-oriented code, discussion now turns to more general styles for designing software architectures.

2.2 Software Architectures

David Garlan and Mary Shaw of Carnegie Mellon University's School of Computer Science teach a course on Software System Architectures, and have recently summarized their knowledge base [Garl93]. This section discusses "the current state of the discipline," by explaining Garlan and Shaw's definition of a software architecture and describing numerous architectural styles.

A software architecture defines the style used to organize a software system. This style helps to structure the flow of control throughout the system, and decides which portions of the system handle which of the required tasks and computations. An architecture also establishes standard techniques used for communicating and accessing data in the system [Garl93,1]. By creating a software architecture, a system becomes easier to grasp for anyone trying to analyze it, design it, implement it, or maintain it. The architecture defines a prescribed structure that the system follows, and comprehending this base structure eases understanding throughout the entire system . While it may not seem that a maintenance programmer has this need, realizing the

scope of the changes she is making within the system can allow her to accomplish her task much more efficiently, responsibly, and safely.

Technically, Garlan and Shaw state that all architectures are broken into *components*, so that the pieces of the system are more understandable and manageable. Communication methods between these components occurs according to the architecture's system of *connectors*. Finally, the behavior of the system must adhere to certain *constraints*, which set rules for combining the components and connectors. The constraints help the system to be more uniform, which also makes it more understandable by decreasing its complexity [Garl93,4-5]. In order to completely understand different methods for defining components, connectors, and constraints, Garlan and Shaw give many examples of architectural styles. This section analyzes those that are pertinent to this research effort, by first explaining them and then discussing their advantages and disadvantages.

The *pipe and filter* architecture is familiar to users of the UNIX operating system because it works very similar to the piping commands found there. A pipe model indicates that data is brought into the system, manipulated or filtered by one component, passed to another component, filtered again, passed in a different form further down the pipe to the next component, passed through another filter, and so on, until the final product is finally achieved. This architectural style has certain advantages. It is highly maintainable, because the different filter components have no knowledge of each other, and any of them can be replaced without affecting the others. This independence also makes the filters very reusable elsewhere, and allows them to be implemented as concurrent processes. The pipe and filter style is also easy to understand, corresponding mathematically to a composition of functions. This

model also has its disadvantages, however. If user interaction is necessary, it does not fit well into this batch-oriented processing scheme. Finally, the pipe passes data down the stream globally, and it must pass all data required at any filter down the entire stream. This model therefore necessitates that the passed data be the union of everything that the different filters need, and it can be costly if they have widely spread data requirements. In summary, the pipe and filter architectural style is simple and neat, but it can require extra overhead and does not wear well in interactive systems [Garl93,5-6].

The second architectural style examined by Garlan and Shaw is the *data abstraction* model. In this model, each component is an instance of an abstract data type, and is termed a *manager* because it is responsible for maintaining its own state, and it keeps this state hidden from other components. Connectors in this style are implemented through subroutine calls, as are common to many block structured languages. Object-oriented architectures are a special case of the data abstraction model, and the advantages of data abstraction have already been described in excoriating detail in Section 2.1. The most significant disadvantage of the data abstraction model is that in order to interact with a component, the identity of that component and its connectors must be known and visible. This problem negatively affects modifiability, because a change in the identity of a component causes a ripple effect throughout the system, as every other component dependent on the changed component must also be altered [Garl93,7-8].

Of special interest to this research effort is the Garlan and Shaw view that inheritance is a method for organizing components, not of connecting them [Garl93,8]. Regardless, a special kind of object-oriented architecture, called an *application framework*, is often used within the object-oriented

community [Str91]. This model provides a base set of abstract or default components within a particular problem area, or *domain*. From this base set, a system developer can use inheritance to derive components necessary for the architecture they foresee for their specific application.

The third architectural style that Garlan and Shaw examine is the *event-based* architecture. This model uses components very similar to the data abstraction model, but these components are connected in an entirely different manner. Whereas the earlier style required explicit calling of another component's subroutines, the event-based model *broadcasts* requests for service throughout the system. The system, responsible for managing which components are interested in which broadcasts, then *implicitly invokes* the necessary components and the data is passed accordingly. The event-based model has its advantages through component independence, as each component is abstract and encapsulated, and can be reusable or concurrent. The main disadvantage is lack of event ordering and determinism, as the system cannot guarantee the sequence in which connections are made, or that they get made within a particular length of time. To correct for this problem, most event-based systems also allow explicit invocation. Another practical disadvantage of this style is the overhead incurred because the system must manage the implicit calls [Garl93,8-9].

The final style that this section looks at in detail is the *layered* architecture. This style organizes its components hierarchically, with each layer providing services to the layers above it, and acting as a client to the layers below it. In the ideal layered system, each layer only has visibility to the layers directly above and below it, effectively encapsulating each layer as the hierarchy extends upward. The connections in a layered model are once again

usually achieved through means of subroutine calls. In addition to reaping the benefits of abstraction and encapsulation, layered models also positively impact modifiability. Because of limited scope, a modification only affects the layers directly connected to the point of change. There are also disadvantages to the layered style. Not all systems lend themselves to a layered breakdown, and those that do are susceptible to high coupling between layers. Efficiency losses can be large when an upper layer must go through multitudes of other layers to get at functionality implemented at a much lower layer [Garl93,9-10].

Garlan and Shaw continue to describe different architectural styles that are used less often in industry [Garl93,10-13]. They conclude their discussion of the different styles by pointing out that most large applications are designed combining more than one style. This intertwining often is accomplished by using an encapsulated hierarchy, with each level's implementation being hidden from, and therefore irrelevant to, the other levels. This *heterogeneous* style takes advantage of each architectural model in the areas in which it is strong, and does not force a style to be used in a situation in which it has shortfalls [Garl93,13].

Having concluded the pedagogical discussion of architectures, discussion now turns to architectures that exist in the real world.

2.3 Current Industrial Simulation Architectures

This section describes software architectures used in the visual simulation industry today. The search for this information is partly obscured, however, because of the proprietary nature of the field. Traditional producers of flight simulators, such as CAE-Link and Evans & Sutherland, rightfully do not care to divulge the details of innovations that give them an advantage in the market-

place. Luckily, they have agreed to share some high level knowledge, and other organizations have published summaries of the software architecture work done at these corporations [Epp94]. The main players in the dissemination of this information are the Software Engineering Institute at Carnegie Mellon University and the Training Systems Program Office of the Air Force's Aeronautical Systems Center (ASC/YT) [Abo93, ASC93, SEI93].

As the primary Air Force agency responsible for the procurement of flight simulators, ASC/YT has years of experience with these complex training devices. It also has quite a stake in ensuring that suppliers can implement the software aspects of flight simulators efficiently and inexpensively. In 1986, ASC/YT realized that simulator software systems were outgrowing their original designs, and that modifications due to changing requirements were becoming more and more impractical. The added complexity also forced simulator vendors to rely on subcontractors, and the resultant geographic and organizational disparity introduced inconsistencies into the development process. In order to correct these problems, ASC/YT began to oversee investigations into the design of a common flight simulator software architecture, or what it termed a *structural model* [Abo93, ASC93,2].

The basic structural model has evolved since its inception, and has recently been adjusted to incorporate the lessons learned from the development of the B-2 and C-17 aircraft systems trainers [ASC93,2]. Boeing's Defense and Space Group has also adopted the structural modeling method, and has started to release results of using the technique in simulations of various fighter aircraft, fire control units, and missile simulation systems [Cri94,280].

Technically, the structural model serves as "a pattern for specifying and implementing software system functionality" [ASC93,5]. The two main as-

pects of ASC/YT's structural model are *partitioning* and *coordination*, and they correspond directly to the components and connections described in Section 2.2. Partitioning refers to the strategy that systems analysts use to divide a complex problem into smaller, less complex, and more manageable pieces. Coordination involves the method by which the partitioned components interact with each other. Establishing strategies for partitioning and coordination encourages consistency and eases integration by giving the numerous members of a development team a common implementation plan from which they can work. Different partitioning and coordination strategies may all lead to proper system functionality, but they may also vary in their attainment of basic software engineering goals like modifiability, reusability, and efficiency. Because these goals are often contradictory, the software systems developers will have to evaluate which goals pertain to their simulator project, and choose the strategy most suitable for their particular needs. ASC/YT calls its software architecture a model because it permits rapid establishment of partial solutions that resemble the desired system. These models can be evaluated inexpensively and adjusted incrementally until the end product is finally achieved [ASC93, 5-11].

Boeing calls its latest incarnation of the structural model the *Domain Architecture for Reuse in Training Systems (DARTS)*. DARTS partitions the problem space of a flight simulator into twelve *segments*, each of which corresponds to a flight simulator subproblem. Examples of these subproblems include flight dynamics, radar, and propulsion. Each segment is further divided into components which represent air vehicle parts or functions. The designers of DARTS consider the coordination aspect of the structural model "of paramount importance." Traditional simulation systems have used global in-

terfacing between components, but coordination must be "defined and controlled" to ensure correct functionality and reliability [Cri94,273]. Because of this belief, DARTS coordinates its segments by using message passing as opposed to shared memory. Although shared memory is generally faster, message passing is less dependent on hardware platforms. Deciding to use a coordination strategy that values reliability and portability over efficiency is the first step in creating a software architecture that can truly be applied in multiple environments [Cri92,2-5].

The techniques of both ASC/YT and Boeing give an architecture that is too complete for any real system. They model every known aspect of the flight simulator domain, so that nothing will be overlooked on any particular flight simulator. Each development team will then have to hone or tailor the architecture to fit its application. This approach is in keeping with the general trend in the field of domain analysis [Cri92,5].

It is surprising that, despite the current development trends, neither the ASC/YT nor the Boeing architecture is overly object-oriented. The Boeing team states, "Note that the analysis that produces the final architecture begins with functional decomposition and ends with what can sensibly be described as objects" [Cri92,5]. The flight simulator systems contractors have basically chosen to exploit the advantages of both the functional and object-oriented programming methodologies. Partitioning can occur at the base level until simple, low-level objects can be designed and implemented. At the higher level, coordination dictates an easily understood flow of control, which is more easily analyzed in a functional manner.

The discussion of background topics now moves from simulation architectures to the Performer graphics programming library.

2.4 The Silicon Graphics IRIS Performer Library

IRIS Performer (Performer) [Coo92, McL92, Har94] is a toolkit that allows graphics programmers to create high performance applications on Silicon Graphics hardware systems. It is built on top of the existing IRIS Graphics Library (GL) [McL91], which eases low-level rendering on SGI platforms. Performer supports two seemingly orthogonal objectives: building applications more easily through a well-defined programmer interface and increasing performance of any GL application. It turns out, however, that since Performer combines optimization with abstract calls to modules, both of these objectives are actually intermingled. Performer therefore greatly enhances the productivity of graphics programmers by allowing them to minimize development time for their applications and automatically maximizing the visual impacts and effects of their efforts.

Although Performer factors out many of the tasks necessary in graphics programming, it does not constrict the application developer's creativity. The dynamics of the visual simulation objects are still left to the developer, who has the freedom to define the entire feel of the project. The basis of the developer's creativity lies in the models she uses to represent the entities in her simulation. These models are usually established by using a three dimensional modeling tool, which stores the geometric representation of an entity in a database. Performer supports many different modeling tools, reading their databases and translating them into data structures which it renders to produce an application. The primary modeling tool used in the AFIT Graphics Lab is called *MultiGen*, and it is developed by Software Systems [Mul92].

Performer's magic is created through the means of two main libraries, the first of which is called *pr*. The code to optimize the visual impact of the

graphics rendering is kept in this low-level library. It contains functions that are vital to concurrent real-time graphics programming, including highly optimized math functions, geometry processing routines, sophisticated state management techniques, memory allocation techniques, and other rendering tricks. Although these concepts are rather simple to fathom, the intricacies needed to make subtle improvements in their performance can be quite complex. Acknowledging this fact, Performer provides abstract calls to these functions, and implements the details itself, allowing the applications developer to concentrate on less mundane activities.

The second Performer library, *pf*, holds the code that allows the applications programmer to easily access visual simulation functions. This library also defines a *rendering tree* data structure for holding the entities being represented in the *scene*, as well as the means to methodically traverse this structure. When told to do so, Performer assimilates the model databases described above into the rendering tree, so that they can be properly displayed in the scene. It is important to note that the pf library totally encompasses the pr library, and because of this structure, the client programmer can access all Performer functionality either directly or indirectly through pf.

Visual simulation applications are implemented by Performer in the pipe and filter manner described in the section above on software architectures. This pipelining allows extremely efficient multiprocessing, exploiting as many processors as are available to increase the computational throughput. Performer continually repeats a three step algorithm that traverses the rendering tree structure and its corresponding scene. The first step in this algorithm, *application*, actually moves the objects in the scene by conducting the necessary calculations, and is defined by the programmer. The second

step, *cull*, moves through the tree deciding which objects are inside the field of vision and therefore need to be drawn. The cull step builds a display list of these objects. The final step, *draw*, renders the objects in the display list. The fact that these three steps are separate makes it possible to split them into threads and take advantage of multiprocessor machines. If this optimization can be accomplished, however, Performer does so automatically and the application programmer need not be concerned with any details.

By default, the complicated and tedious cull and draw processes are accomplished automatically by Performer. However, an application developer may customize these processes through an established method of *callbacks*. The programmer provides his own functions for culling and drawing, and notifies Performer of their existence during Performer initialization. During the simulation, Performer calls his functions at the appropriate time, and the customization occurs. This Performer feature is useful if certain entities within the simulation must react to input devices, which are usually read on the draw thread, or have specialized output requirements. Callbacks are also used by Performer and GL to perform window management.

Finally, Performer has some other important features which application developers can use. *Channels* can be used by the programmer to set up different views into a scene, as if various observers were watching from completely different angles. Multiple views are also useful if entirely different representations of a scene are necessary. For example, when simulating a radar in a plane's cockpit, both the radar screen and the canopy provide views of what is occurring outside the plane, but these two "windows" display the scene in utterly distinct fashions. In addition to multiple view handling, Performer provides mechanisms for using shared memory to enhance processing

speed through the pipes. This feature allows developers to hone the performance of their applications. Other features in Performer allow the developer to include such features as collision detection, sequenced animations, atmospheric effects like fog or haze, or light sources to imitate beacons, stars, and sunlight.

This section has provided a brief overview of Performer and its capabilities. More complete coverage can be found in the documentation for Performer and its related tools [Coo92, Har94, McL91, Mul92]. Discussion now focuses on ObjectSim, the architecture that was created in the AFIT Graphics Lab by Mark Snyder to provide further abstraction on top of Performer.

2.5 ObjectSim Concepts

Since 1989, the AFIT Graphics Lab has been sponsored by the Advanced Research Projects Agency (ARPA) to investigate low-cost distributed interactive simulations. As technology has progressed since then, so have the capabilities of the applications produced by the score of graduate students who have worked in the Lab. However, this increase in capability has been accompanied by a corresponding increase in software complexity. In the academic cycles ending in 1991 and 1992, it became apparent that the students were spending more time redoing tasks common to all visual simulations than developing solutions for their particular projects. Patricia Lawlis, Assistant Professor of Software Engineering at AFIT, became involved in the research in 1992 to attempt to manage a reuse effort within the Lab. Reuse would allow the students to concentrate more of their efforts on their specific simulations, instead of recreating common graphics functionality. Lawlis enlisted the help of her student, Mark Snyder, and he produced the ObjectSim framework to curb the

complexity problems in the Lab. This section outlines the basic design of ObjectSim [Sny93].

Snyder's primary task was to factor out the commonalities between the four ongoing projects within the Graphics Lab [Eri93, Gard93, Ger93, Kun93, Sol93, Wil93]. He began his effort by examining components that already existed, to see if they could be reused. His survey quickly revealed that these components, developed for specific applications, were not malleable enough to use on varied projects. Snyder realized that it would be more productive to make reusable components from scratch [Sny93,31-32]. Of notable exception were the network communications components, which were reused. The probable reason for this exception is that the original designer of these components, Steven Sheasby, has been maintaining them since their inception [She92, She94]. He could therefore serve as living documentation to define their use, whereas the knowledge necessary to use the other components had disappeared with the graduation of their designers. The network communication components were never completely integrated into ObjectSim, however.

After determining what he could reuse, Snyder performed a requirements analysis by talking with the other students in the Graphics Lab. Through this analysis he was able to establish the functionality that was common to every application [Sny93,32-33]. Snyder decided to allow the other students to access these common capabilities by means of an application framework. This programming paradigm provides an avenue for software reuse, but at the domain level instead of the traditional component level. Within the domain, the components of the framework provide templates from which actual components can be derived and customized. The framework therefore "provides major savings," as the basic architecture for the code already exists

[Sny93,39-40]. Not surprisingly, components of an application framework are often implemented as abstract classes in object-oriented programming languages.

Snyder began to rapidly develop ObjectSim by incorporating components donated by the other six students. He also developed generalized implementations of common components on his own. Snyder designed test applications to evaluate new functionality, and through this methodology he was able to quickly coerce the Virtual Cockpit to rely on the ObjectSim framework. The other applications followed soon thereafter [Sny93,40-44].

In terms of architectural style, ObjectSim belongs to the data abstraction model. Its object-orientation defines abstract interfaces to different component classes, with straight function calls acting as connectors. The use of abstract classes determines the basic architectural structure, and forces the application developer to adhere to the provided scheme for components and connections in order to benefit from the framework. While ObjectSim itself adheres to the data abstraction model, its use of the pipe and filter Performer architecture makes the entire system heterogeneous. The fact that Performer is kept separate from ObjectSim, with interaction again occurring by function calls, adds layering into the system as well. Under Performer are two lower layers, one for GL and one for the IRIX operating system, but they exist at a level of abstraction below what is examined in this thesis. Figure 14 shows the architectural layering of ObjectSim.

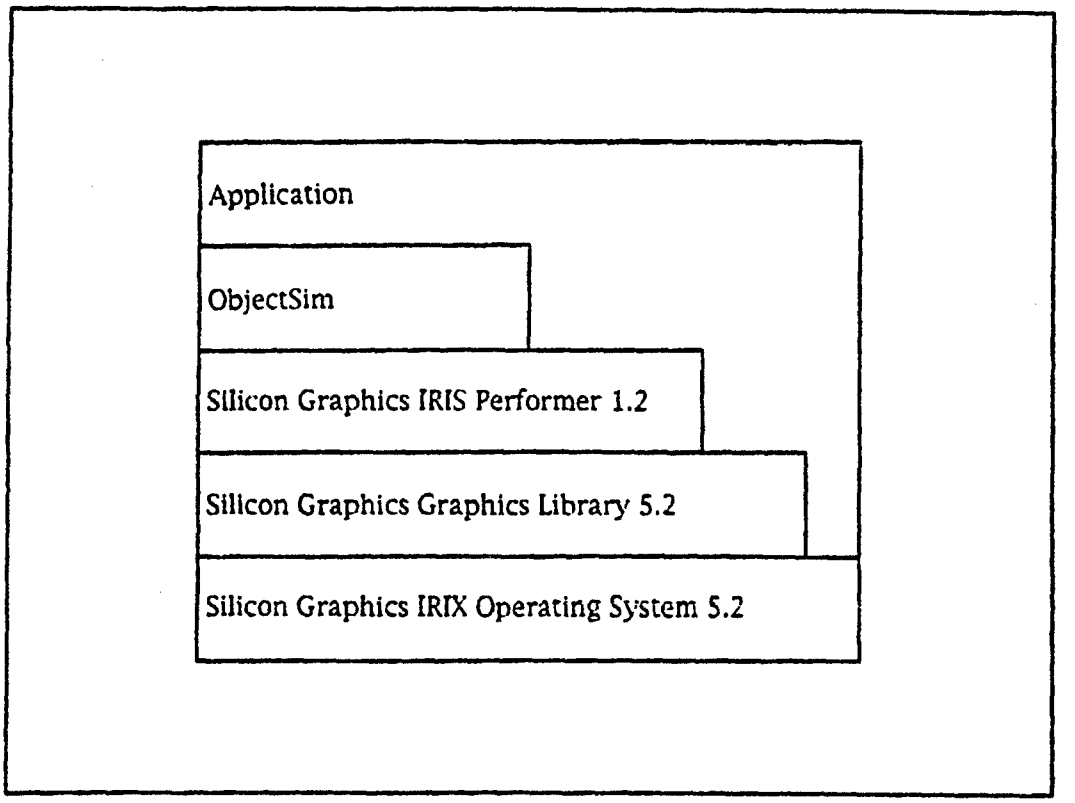


Figure 14. ObjectSim Architectural Layering

Figure 15 shows the ObjectSim Object Model. The main objects in the application framework and the relations between them are indicated in the Rumbaugh diagram. The most basic object is the Flt_Model, which stores database information describing how an object appears graphically. A Terrain serves as the visual background for an application. A Simulation serves as the basis for an application and encompasses a Terrain and any number of Players, the entities whose interaction define the graphics application. A View allows vision into the application's scene, and a Modifier can be used to move a View around the scene. Finally, the Pfmr_Renderer controls the actual drawing by making the necessary Performer calls.

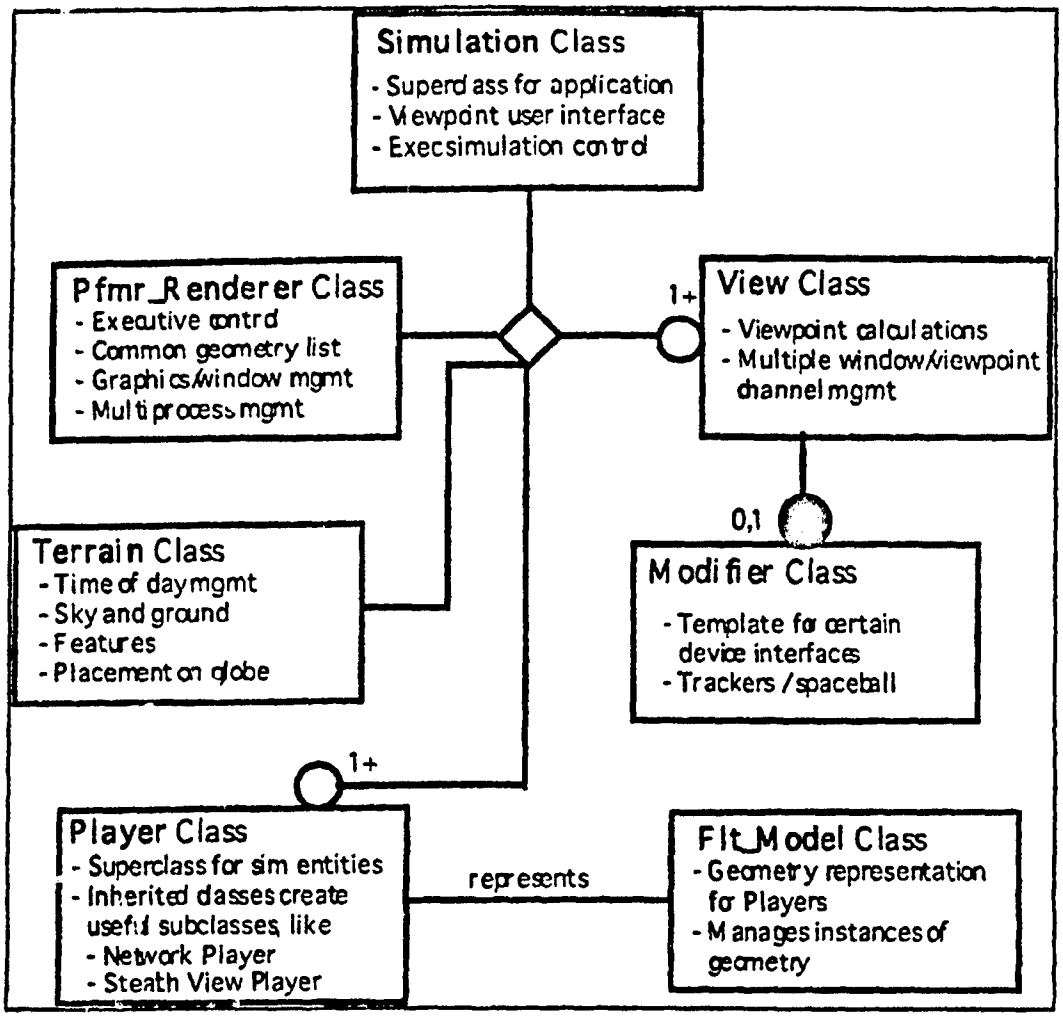


Figure 15. ObjectSim Object Model [Sny93,45]

The Terrain, Simulation, Player, and View classes of the ObjectSim framework are abstract. This approach allows application developers to customize their simulations freely while letting them exploit proven methods and algorithms. It also encourages developers to produce standardized applications, making them more interoperable and modifiable. Two of the concrete classes were not intended to be portable, as their names reflect. Flt_Model

algorithms. It also encourages developers to produce standardized applications, making them more interoperable and modifiable. Two of the concrete classes were not intended to be portable, as their names reflect. Flt_Model handles *Flight* format models created using the MultiGen visual database development tool, and the Pfmr_Renderer is intended to isolate the Simulation's dependency on the Performer library.

While this summary has simplified ObjectSim in many ways, it does convey the general ideas behind its development. ObjectSim was an unqualified success in the AFIT Graphics Lab, increasing student productivity between thirty and forty percent [Sny93,79]. The next chapter looks at ObjectSim more closely in order to develop its successor, Easy_Sim.

III Architectural Design

This chapter relates the analysis performed on the ObjectSim architecture and the lessons that were learned from it and applied to the formation of *Easy_Sim*, a language independent software architecture for visual simulation systems. This chapter firsts analyzes the advantages and disadvantages of the method Mark Snyder used to develop ObjectSim, and then describes the radically different method used for Easy_Sim. The majority of this chapter describes the Easy_Sim classes and their relationships. This task is accomplished by examining each component of ObjectSim, retaining its positive points, and modifying its negative aspects. The chapter concludes by presenting the final Easy_Sim Object Model, and examining the overall design of the Easy_Sim architecture.

3.1 The Design Processes

Mark Snyder developed ObjectSim with four large simulation projects and six other students reliant on his success. This pressure and dependence had both positive and negative effects on his design efforts. This section describes how Easy_Sim benefited from ObjectSim's achievement while trying to avoid its pitfalls. It first examines the method used for developing ObjectSim, then outlines the diametrically opposed approach used with Easy_Sim.

3.1.1 The ObjectSim Development Method

Mark Snyder created ObjectSim according to what he termed the *necessity model* [Sny93,44]. When the Graphics Lab projects were originally converted to use ObjectSim, Snyder analyzed their components for inclusion in the framework. Classes deemed suitable were generalized and incorporated into

ObjectSim. Alternatively, if a student desired new functionality in his application, he would implement a solution locally, and Snyder would evaluate the utility of the new code for everyone, once again incorporating it if it promised to "promote the general welfare."

Snyder's necessity model allowed a tremendous number of ideas, designs and components to be considered for ObjectSim, and enabled them to be considered early in the thesis life cycle. The amount of useful code in the Lab was able to expand rapidly because of this technique. More importantly, the code brought into ObjectSim was tested in numerous applications, so any defects were rapidly detected and eliminated.

While the necessity model allowed the general productivity in the Graphics Lab to increase dramatically, it also had some negative effects. Because of the dynamic nature of the framework, the other students often had to suspend their own projects to incorporate new versions of ObjectSim. Configuration management problems occurred frequently [Sny93,ii]. With seven individuals contributing code, conflicts in style and convention were common. Unfortunately, these conflicts were carried into the framework, and because the changes needed to be made quickly, standardization was often performed hastily. Many of ObjectSim's component and connector names remain inconsistent. The haste associated with version updates also had a more long lasting side effect--it created an oral culture within the Lab. Modifications were suggested, made, and forgotten before their rationale was documented. ObjectSim is currently filled with enigmatic code that seemingly does nothing, but whose removal causes utter destruction.

The reader may have noticed a shift during discussion of the necessity model. It lowered the focus of ObjectSim's development from the analysis and

design level to the code level. Unfortunately, the necessity model was often reactive when it should have been guiding, letting the implementation drive the design. This shift is evident when ObjectSim is evaluated thoroughly, especially in the area of network interaction. Because the Graphics Lab depended on a contractor to handle this area of their projects, network interaction capabilities were never completely incorporated into ObjectSim. This divergence resulted in network components that handled interaction differently for each application. Anytime new functionality was added, kludges had to be introduced in many places, including the basic ObjectSim framework [Sny93,57, She94]. The intermingling of implementation and design is also evident in Snyder's thesis, as his design chapter frequently refers to implementation issues [Sny93,Ch4].

3.1.2 The Easy_Sim Development Method

Easy_Sim was developed without the pressure of any other students reliant on its results, as the five students working concurrently used the existing ObjectSim framework. This decision was made consciously because of the risk involved with the primary implementation of Easy_Sim occurring with an untested language in an untested environment. Accordingly, Easy_Sim was designed with a set of circumstances completely reversed from the ObjectSim case. This section briefly describes those differences, and concludes by outlining design decisions common to all Easy_Sim classes.

Where the production of ObjectSim benefited immensely from having seven developers simultaneously recognizing requirements, contributing designs, and testing solutions, Easy_Sim is primarily implemented as a solo effort. Its requirements are drawn by trying to match ObjectSim functionality and

conversing with the current students, but no code has been donated through this method. The Easy_Sim implementation draws some ideas and code from a demonstration program, *Paintball*, under development at Silicon Graphics by Wes Embry and John Templeton [Emb94]. This effort converts an Ada 83 program to Ada 9X, but does not exploit the object-oriented capabilities of the new language. Paintball's contributions to the Easy_Sim architectural design are therefore limited. Consequently, Easy_Sim has been developed much more slowly and with much less implementation testing than ObjectSim, but with considerations for it to be language independent.

The development method for Easy_Sim does have its advantages, however. Because the design is not influenced by the need to be rapidly integrated, it remains more pure. Its development by one individual also makes the components and connectors more standardized, and configuration management is not a factor. Finally, design decisions are documented both in this document and in the code, so that continuing work both at AFIT and elsewhere will be able to understand Easy_Sim's evolution.

This section has analyzed the positive and negative aspects of the ObjectSim and Easy_Sim architectural design methods. Before the design of each class is detailed, the next section outlines the overall conventions used throughout the entire Easy_Sim design.

3.2 Easy_Sim Class Design Conventions

The classes in Easy_Sim are designed using the design level ideas of the ROMAN-9X technique (see Section 2.1.4). This method keeps the different classes consistent by providing standardized naming conventions and constructs, and it enhances the definition of the data abstraction architecture's

components and connectors (see Section 2.2). Each class has default initialization and finalization operations, a parameterized initialization operation called *Configure*, and Get and Set operations as appropriate for its attributes. Classes that contain visual models have a special Get operation, *Image*, which returns the node in the rendering tree corresponding to the root of the object's graphical representation. The existence of the Get and Set operations is assumed in the remainder of this chapter, and no further mention of them is made. Most classes also have an *Update* operation, which determines the behavior of an instance of that class, and a *Draw* operation, which defines any necessary class specific rendering functionality. Both the Update and Draw operations are called each frame of the simulation.

Resources needed commonly throughout the Easy_Sim architecture are defined in a common location. The most visible of these resources is *Coords*, the data type used to describe coordinates in an application. This type contains the *X*, *Y*, and *Z* vector values which define the *Position* where an entity exists in the simulation. Coords also contains a vector referring to the entity's *Direction* at its Position. This vector comprises three values as well: the *Heading*, *Pitch*, and *Roll* of the entity.

The rest of this chapter examines the design of each component within ObjectSim, examines any positive and negative aspects of that design, and details how that component has been migrated into Easy_Sim. Each section is concluded with a Rumbaugh diagram showing the two versions of the class next to each other so that they can be easily compared. The entire ObjectSim Object Model [Sny93,45] is presented as Figure 15 at the end of Chapter II. Discussion begins with the most basic classes and progresses through the more

complex components. The discussion of new Easy_Sim components is interspersed as appropriate.

3.3 The Flt_Model Class

The intent of ObjectSim's Flt_Model class is to provide the abstract capability of handling the geometric models used to represent images in a visual simulation. The class name is prefixed by "Flt" to show that it can only represent models created in MultiGen's Flight format [Sny94] (see Section 2.4). The main operation of this class, *readmodel*, reads in a Flight format database, converting it into a format that the simulation can process. This conversion is handled by a simple Performer operation, and it stores the database into an attribute, *root*. This attribute forms the base node of the tree representing the geometry for the Model. Flt_Model also contains a constructor for initialization purposes. Figure 16 shows the Rumbaugh diagram for Flt_Model.

Since the completion of ObjectSim, a new version of Performer has been released [Har94]. This version contains more flexible conversions of geometry databases, enabling this class to become independent of the Flight format. This generalization is the first change made to this component in Easy_Sim, and the component is renamed *Model* to reflect its added capabilities. The new architecture also allows the reading of the Model database to be interpreted as a form of parameterized initialization, and Easy_Sim therefore assigns this functionality to Model's Configure operation.

In its readmodel operation, ObjectSim's Flt_Model class provides the ability to copy a model. This characteristic enables a simulation to avoid inefficient replication of complex databases if two or more entities share a common geometric representation. For example, in an airfield simulation there

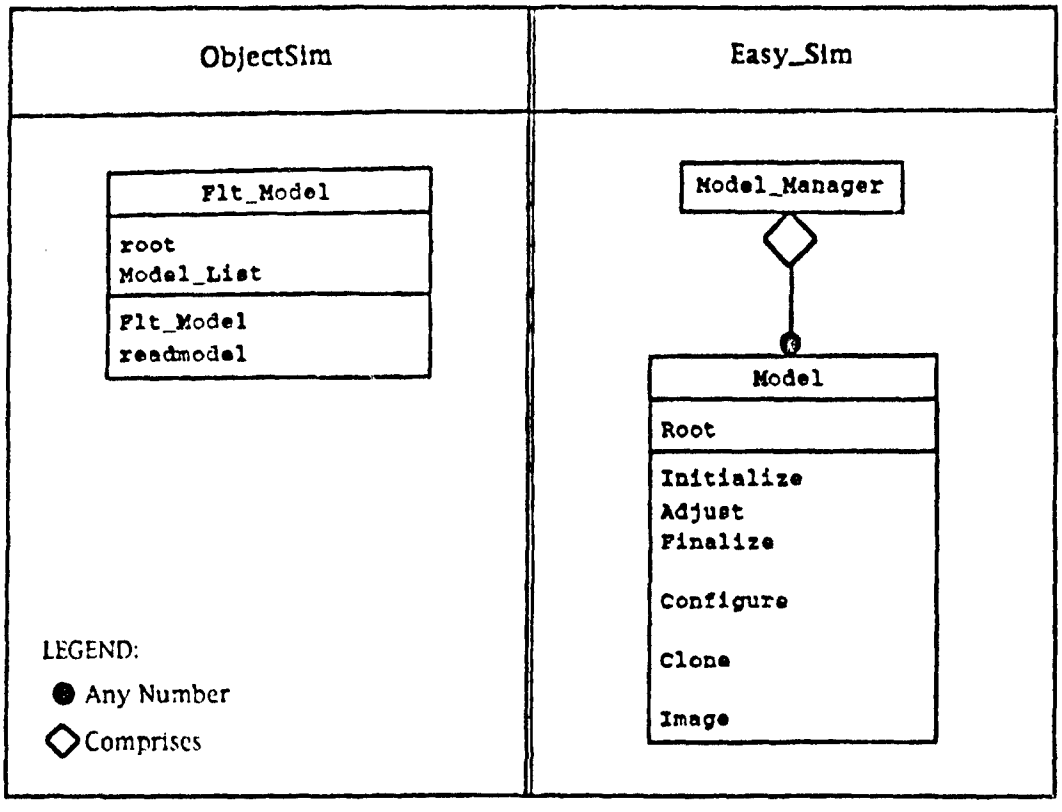


Figure 16. Model Class Object Model Diagrams

may be many similar F-15s flying in the area, but one Model of an F-15 can represent all these planes. ObjectSim's design of this feature uses a straight duplication technique, which works adequately for simple models, but fails on models with moving parts. At the airport, an arriving plane would have its landing gear down and visible while a departing or circling plane would have its landing gear up and hidden from view. ObjectSim's omission is corrected in the Easy_Sim Model class, and is available publicly by means of the *Clone* operation.

Finally, the ObjectSim design of Flt_Model maintains a list of all the models to be used in an application. In Easy_Sim, the Model class is simplified

to operate on a single Model, and a *Model_Manager* class is introduced to track multiple instances of the Model class. This change makes the Model class consistent with other Easy_Sim classes and is explained further in Section 3.7. Figure 16 shows a Rumbaugh diagram representing the Easy_Sim Model class.

3.4 The Terrain Class

The abstract ObjectSim Terrain class is intended to provide a template so that each application can develop a unique visual background in which its Simulation can operate. Features intended to be implemented here include lighting models, weather models, time of day management, the terrain, and any other environmental variables. ObjectSim comes with a default Terrain subclass, *Simple_Terrain*, which uses simple sun and horizon models as well as a Flt_Model to represent the ground. Figure 17 diagrams the ObjectSim Terrain and Simple_Terrain classes.

The Terrain class is a perfect indicator of the disadvantages of ObjectSim's necessity model development method. Originally part of the Virtual Cockpit application [Erl93, Ger93] and incorporated into ObjectSim, the description of Terrain does not fit the Space Modeler [Van94] application, which uses stars and planets as its background. The ObjectSim Object Model also does not reflect the relationship between Terrain and Flt_Model, where Terrain is mysteriously implemented as a derived subclass of Flt_Model. This kludge allows a Flt_Model to be used to represent the Terrain, without explicitly making it an attribute.

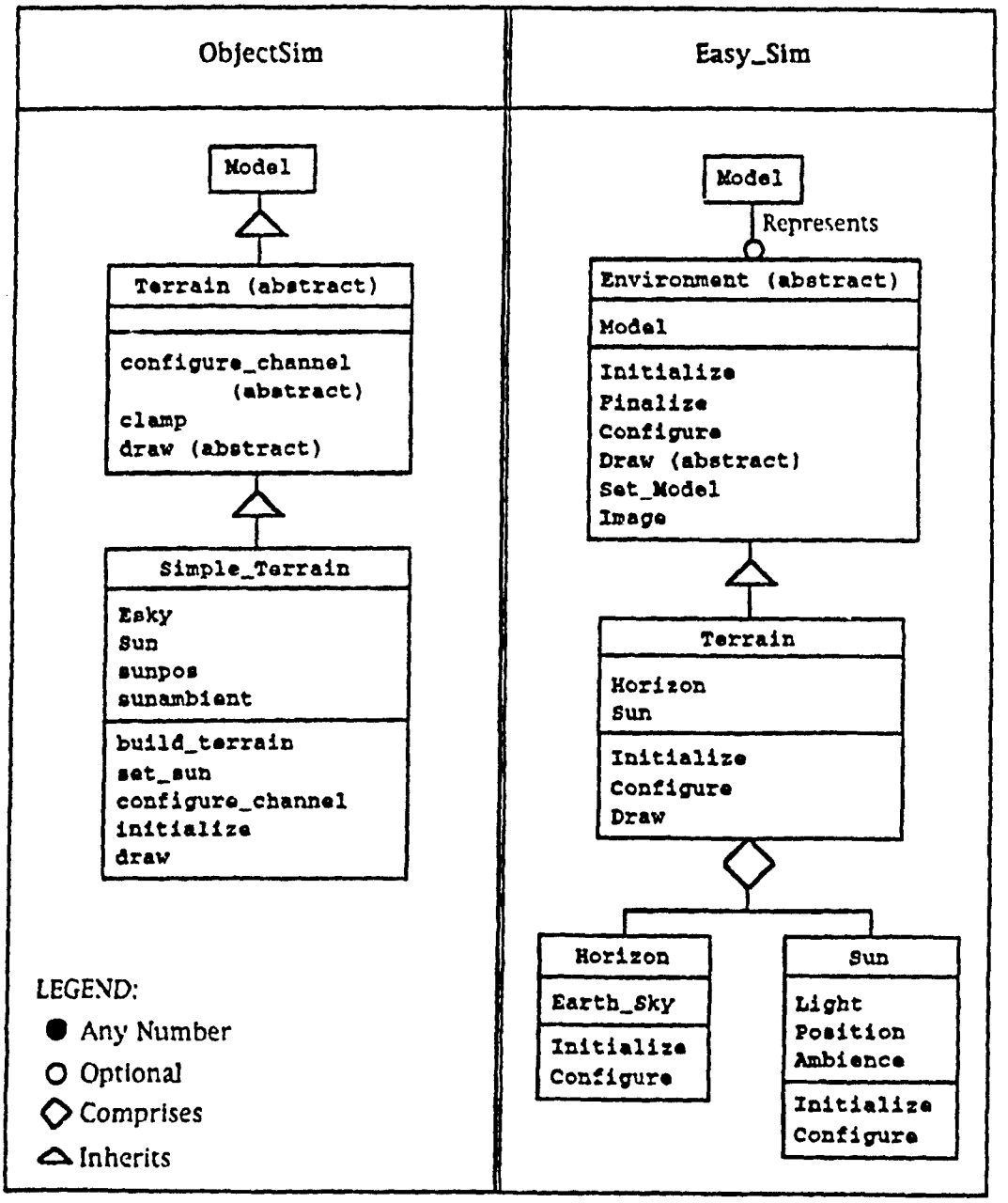


Figure 17. Environment Class Object Model Diagrams

Fortunately, ObjectSim's Terrain class also indicates the advantages of Snyder's necessity model. The creation of an abstract class clearly defines the

component and connectors that a developer needs to form a useful environment, and illustrates an example of the object-oriented data abstraction architectural style described by Garlan and Shaw [Garl93,7-8] (see Section 2.2). The Terrain class has two initialization operations, *clamp* and *configure_channel*, and an abstract *draw* operation. Simple_Terrain adds two more initialization operations, *build_terrain* and *initialize*, along with attributes to represent the horizon and sun.

The use of abstract classes is adopted by Easy_Sim for this class and throughout many of the other classes in the architecture. Easy_Sim renames the Terrain class *Environment* to indicate its more flexible capabilities and prominently shows the non-inheritance based relationship between the Environment and Model classes. Easy_Sim assigns the functionality of the initialization operations to *Initialize* and *Configure*, its default and parameterized initialization operations, as appropriate. Easy_Sim retains the abstract *Draw* operation.

The new architecture also renames the default Simple_Terrain *Terrain*, for those applications where the basic ground, sun, and horizon model is sensible. Easy_Sim breaks its Terrain class into smaller classes, however, providing functionality for *Sun* and *Horizon* classes separately, and making Terrain an aggregate containing these classes. This modular approach establishes a precedent, whereby a subclass can gather functionality from many smaller classes and organize this functionality in one place to suit the requirements of the architecture. This method makes building Environments much more flexible and establishes a set of modules that are reusable both at the design and code levels. Figure 17 shows a Rumbaugh diagram of the Environment classes.

3.5 The Player Class

ObjectSim's abstract Player class serves as the base class for defining the entities whose interaction defines the simulation. The Player class follows the same basic approach as Terrain, establishing an architectural template for a component and its connectors. Each Player has an attribute which defines coordinates for its position and direction within the simulation, and three operations modify this attribute. One operation, *move_along_heading*, advances the Player along its current path a given distance, while a second operation, *look_at_point*, pivots the Player to face a given position within the simulation. The *propagate* operation is abstract, and it is the avenue through which a subclass defines how the Player behaves during each advancing frame of the simulation. The Player class also provides an abstract initialization operation, *init*.

Most ObjectSim Players contain a Flt_Model which represents the Player's physical appearance. Some Players may exist in a simulation without representation, however, purely existing to provide a vantage point into the scene. ObjectSim categorizes these Players by calling them *Stealth_Players*, and they are generally derived from another abstract Player subclass called *Attachable_Player*. This class is used for players to which a View can be attached, and it contains attributes relating the position and direction of the View relative to the Player. Because most Players within an ObjectSim application can view the scene, simulations derive the majority of their Players from the Attachable_Player subclass. Figure 18 outlines the ObjectSim Player classes.

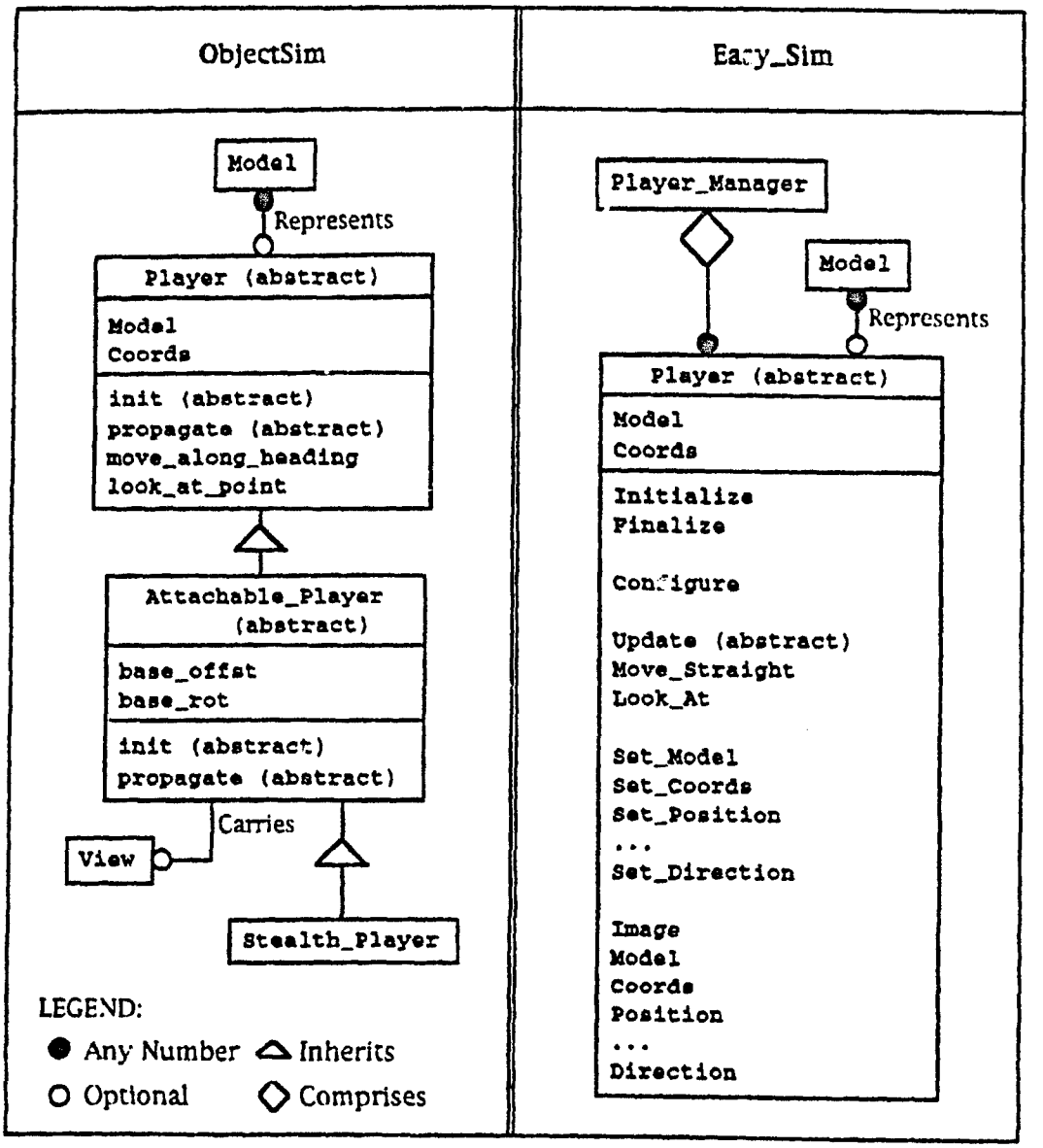


Figure 18. Player Class Object Model Diagrams

Easy_Sim uses the sound design of ObjectSim's abstract *Player* class in defining it own *Player* class with only minor cosmetic changes. The *propagate* operation is equated with Easy_Sim's standard *Update* operation. While some other ObjectSim classes also use the name *propagate* to denote the operation

that updates an instance of the class each simulation frame, this similar operation is named inconsistently throughout the ObjectSim architecture. Easy_Sim uses *Update* throughout all of its classes to provide consistent connection protocols within the architecture. Easy_Sim also renames the ObjectSim Player's move_along_heading and look_at_point operations, calling them *Move_Straight* and *Look_At* respectively. Finally, Easy_Sim converts the init operation so that its functionality is achieved by the controlled *Initialize* constructor.

The Attachable_Player class is not carried into Easy_Sim. The purpose of this class in ObjectSim is to track the position and orientation of the View that is attached to the Player. The Easy_Sim interpretation of these View attributes places them within the View class. Easy_Sim application developers are free to develop subclasses similar to ObjectSim's Stealth_Player concept by deriving directly from the Player class.

ObjectSim makes no provision for standard protocols for handling multiple Players. Easy_Sim introduces the *Player_Manager* class to supply this added functionality, and this class is further described in Section 3.7.

Figure 18 shows a Rumbaugh Object Model diagram of the Player classes. Some of the Get and Set operations for the Coords attribute have been omitted for brevity, and are listed in full with the Modifier class in Figure 20.

3.6 The View and Modifier Classes

The intent of ObjectSim's View class is to allow the users of a visual simulation to look into the scene being rendered. Unfortunately, the View class is also a victim of the negative aspects of the necessity development model, and description beyond its purpose is rather clouded. A View must be related to an

Attachable_Player with its *attached* attribute, but this connection is not shown in the ObjectSim Object Model. At one point the View class is described as encapsulating a single view [Sny93,48], while the Object Model diagram labels it as managing multiple views. Examination of the implementation shows the former to be true. ObjectSim's View class operations include *setview*, which updates the Views each frame, and *attach_to_player*, which switches a View between different Players. The View class also provides two separate initialization operations, *alloc_shared* and *new_view*.

To quickly clear a common misconception, a View can be attached to different players consecutively, and a scene can therefore be observed from many different angles in a single View simulation. Multiple Views are only needed when different aspects of the simulation need to be observed simultaneously. Some examples include a rear view mirror in a driving simulation, a radar screen in a flying simulation, or an inset channel box on a television set. This last example demonstrates the case where the different Views may not be observing the same scene. In order to manage this complexity, each ObjectSim View has two Performer related attributes, *chan* and *scene*. A Performer Channel represents a window in a simulation, while a Scene serves as the root of that Channel's rendering tree, storing the graphical data seen in each window.

The ObjectSim View class also provides operations which the application developer can use to customize Performer's cull and draw processes (see Section 2.4). Both the *cull* and *draw* operations provide default behavior for each window in the simulation, basically doing nothing. They do, however, define points in the architecture where the application developer can modify this behavior. Customized culling can enhance the application's performance, and

the draw operation can add extra information to the scene, such as text overlays. Figure 19 shows an Object Model diagram of the ObjectSim View class.

In ObjectSim, a View may be associated with a Modifier class, which is represented by the View's *Delta* attribute. The Modifier is used to manipulate the View's position and direction relative to its Attachable_Player, and the Modifier contains coordinate attributes, called *State*, accordingly. Like the Terrain and Player, the Modifier class defines an abstract component with connectors that must be provided in its subclasses. Examples of Modifier subclasses used in ObjectSim applications are the mouse, keyboard, spaceball, and head mounted display. The main operation of the Modifier class is called *poll*, and subclasses must override this operation to define how the device receives input data every frame of the simulation. The Modifier class also provides a *reset* operation and two initialization operations, *init_state* and *init*. Figure 20 displays the Rumbaugh diagram representing the Modifier class.

Easy_Sim does not follow the ObjectSim design for its *View* class. To be consistent with its Model and Player classes, Easy_Sim defines the class to manage a single View, pushing multiple View administration to the *View_Manager* class as described in the next section. As attributes, the Easy_Sim View class incorporates the *Player* to which the View is attached, the optional *Modifier*, and the *Coords* eliminated previously by the loss of the Attachable_Player class. Additionally, the View contains a *Channel* and *Scene*. These attributes allow simulations with multiple Views to display distinct collections of entities in each of its windows.

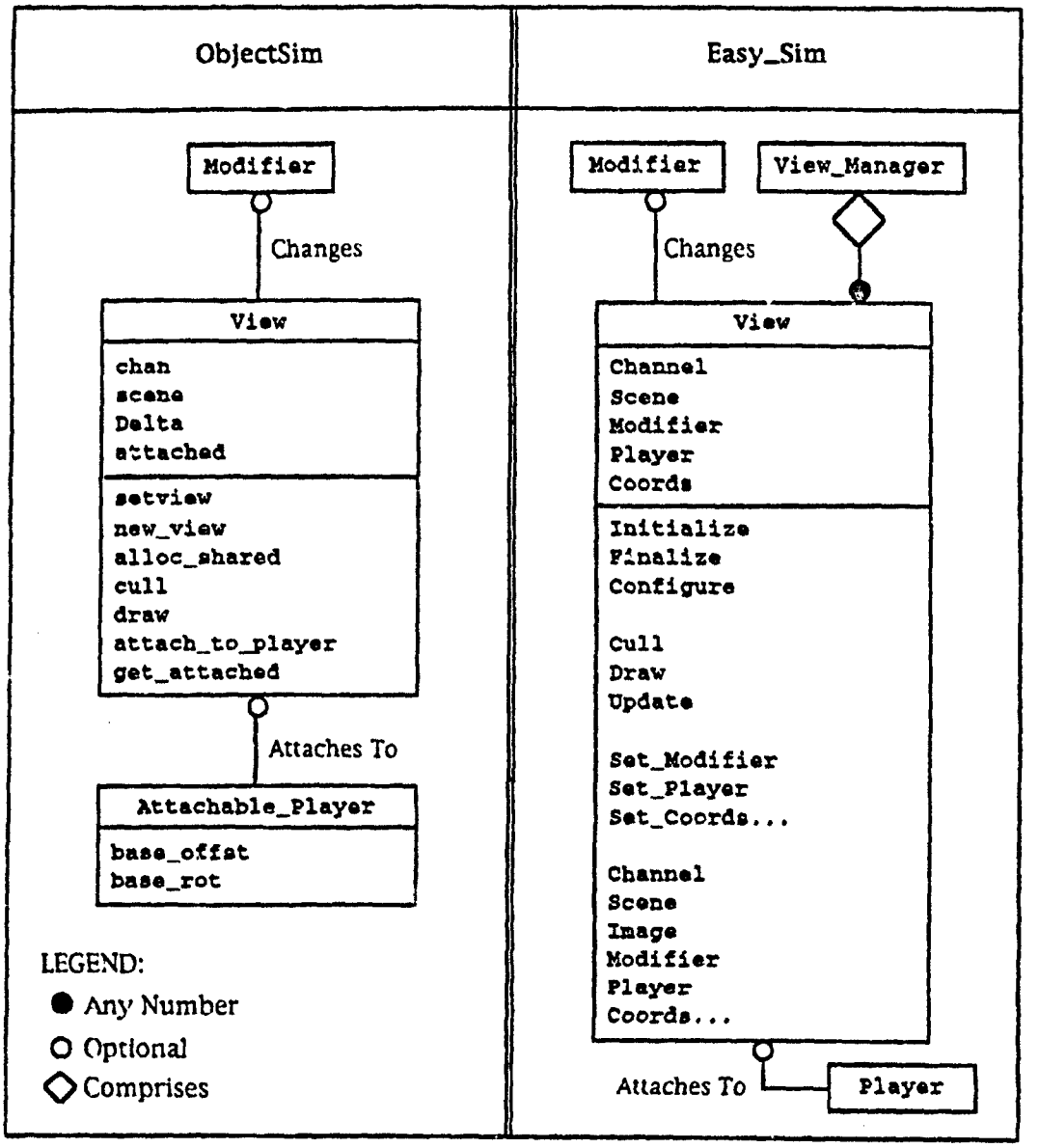


Figure 19. View Class Object Model Diagrams

The Easy_Sim *View class provides* operations equivalent to those present in ObjectSim, but setview is renamed *Update*, alloc_shared and new_view are done by *Initialize* and *Configure*, and attach_to_player is a simple attribute Set

operation. Figure 19 shows the Easy_Sim design of the View classes. As with the Easy_Sim Player diagram, the Get and Set operations on the Coords attribute are not shown, and a full listing can be found with the Modifier in Figure 20.

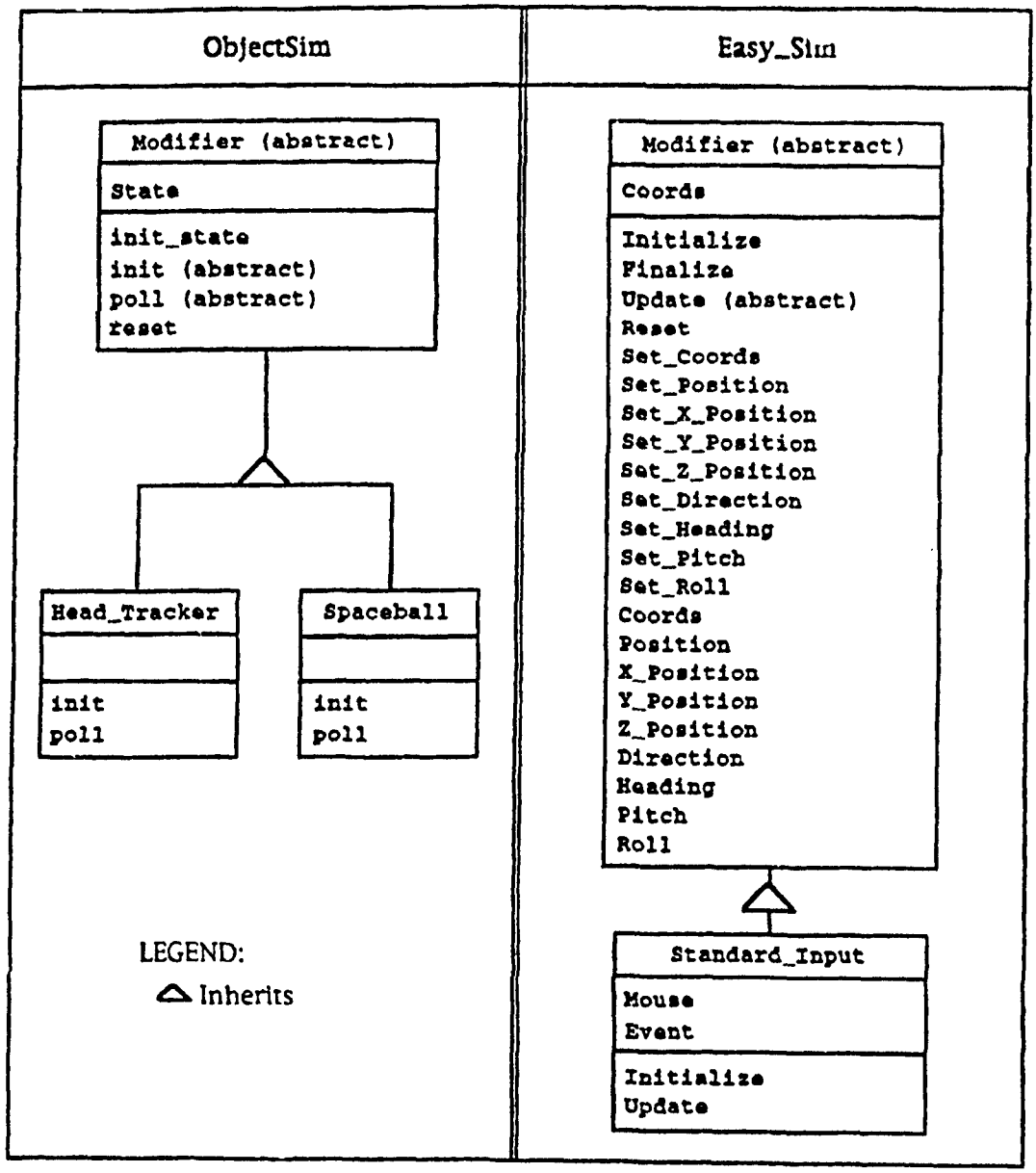


Figure 20. Modifier Class Object Model Diagrams

Easy_Sim reuses the clean design of the abstract *Modifier* class, but renames the poll operation, observing that many input devices use queueing instead of polling to gather data. For this operation, Easy_Sim once again uses its ubiquitous name, *Update*. The functionality of ObjectSim's init_state, init, and reset operations are migrated to Easy_Sim by the *Initialize* constructor and *Reset* operation. The Modifier class works analogously to the Environment class, whereby smaller pieces may be combined in an aggregate that follows the architectural design established by the abstract base class. Easy_Sim provides a default Modifier class, *Standard_Input*, that accepts input from a mouse and keyboard. Figure 20 outlines the Easy_Sim design of the Modifier classes.

3.7 The New Manager Classes

As previously mentioned, the Model, Player, and View classes in Easy_Sim each form an abstraction representing a single object. However, multiple instances of each of these classes are necessary to achieve a viable simulation. ObjectSim handles these multiple instances inconsistently throughout the different classes, as the Flt_Model class administers numerous objects, but the Player and View classes handle only one. Easy_Sim seeks to make multiple instance handling more architecturally harmonious, and achieves this with the addition of container classes to manage multiple Models, Players, and Views. The Easy_Sim architecture benefits from the addition of these classes, as they provide customization points for application developers that are not available in ObjectSim.

The *Model_Manager* class provides an architectural vehicle for controlling the various Model classes that represent entities in the simulation. This manager class defines a simple default operation, *Assign_Model*, which

determines whether a Model already exists and should be cloned to efficiently manage memory. Figure 21 contains a Rumbaugh diagram of the Easy_Sim Model_Manager class.

This simple default design of the Model_Manager should suffice for most applications, but others may wish to customize how the Models in the simula-

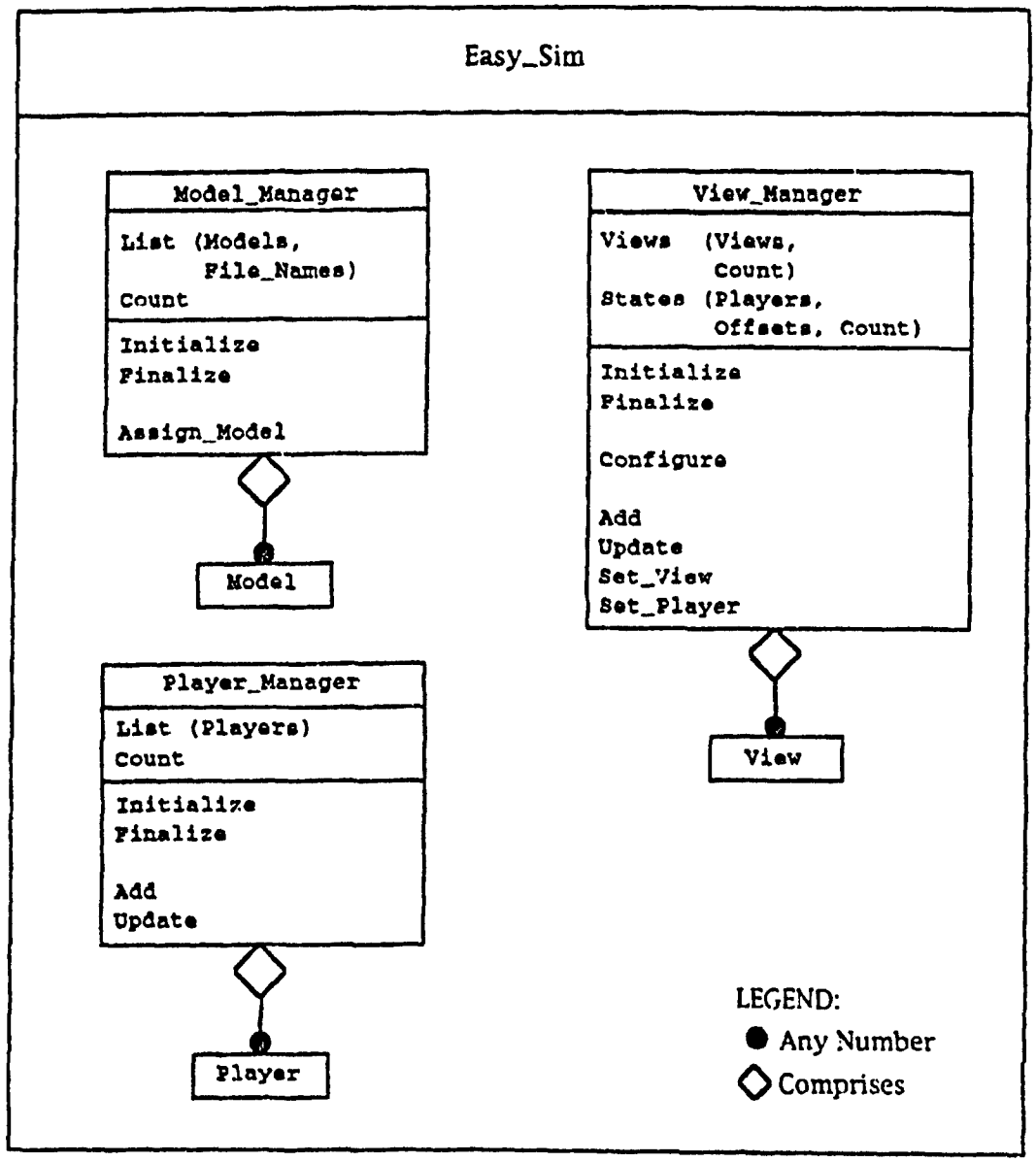


Figure 21. Manager Class Object Model Diagrams

tion are manipulated. A Model_Manager subclass may wish to define different techniques for level of detail control or for compatibility with a different coordinate system. A subclass may also simply require anoth avenue for assigning the Models. The ability to easily customize the control over Models is not present in ObjectSim, so the addition of the manager class into the Easy_Sim architecture has had an impact on the overall design larger than the simple addition of functionality. The addition of this class has created the potential for the addition of many functions by supplying a common architectural component and standard connectors from which customization can readily occur.

The *Player_Manager* class also provides a simple default class, and it maintains the list of Players that interact in the simulation. Its operations include *Add*, which places a new Player into the list, and *Update*, which simply calls the Update for each Player in the list. Figure 21 also shows the Rumbaugh diagram representing the Player_Manager class.

The Player_Manager class also forms the architectural entry point for a limitless number of customizations for Easy_Sim applications. The default class provides no real organization of the Players, but different subclasses could institute methods for optimizing rendering by spatially organizing the Players [Har94,130]. An application could also add collision detection into Easy_Sim applications through the Player_Manager. Most importantly for the AFIT Graphics Lab, the Player_Manager class provides a sound starting point for bringing network player managers into the architecture. Ideally, an abstract class could be designed that is general enough for any distributed interactive simulation. Subclasses could be derived for each particular application, cus-

tomizing its specific needs according to a standard, understandable architectural plan.

Like the other manager classes, the *View_Manager* class maintains a list allowing a simulation to have multiple Views. The View_Manager class also stores a list of *View states*, each of which describes a Player and the relative offset of the last View attached to that Player. Administering a list of View states allows a View to attach to one Player at a particular offset for some time period, then attach to a different Player for another time period, and then eventually reattach to the first Player, remembering the previous offset and orientation. This feature works more easily in ObjectSim due to each Attachable_Player storing its own state, but the elimination of the Attachable_Player class to maintain a pure design (see Section 3.5) forces Easy_Sim to develop this alternate solution.

The View_Manager class provides some basic operations. *Add* places a new View into the list, *Set_View* activates a View, *Set_Player* switches the Player to which a View is attached, and *Update* calls the Update operations for all active Views. View state management is undertaken in the Set_Player operation. Figure 21 shows a Rumbaugh diagram of the View_Manager class.

Like the other manager classes, Easy_Sim's View_Manager class provides default functionality, fully expecting that applications will customize their View management by inheriting from the component and connector protocols established by this class. Window management will probably be the most widely used reason for tailoring the View_Manager class in applications, as each View has its own window in the display. The View_Manager class will also administer input device handling, because user input is commonly obtained from windows through the operating system.

3.8 The Pfmr_Renderer and Simulation Classes

The main purpose of the ObjectSim Simulation class is to glue together all of the other pieces within the simulation. Like many other classes, the Simulation class is abstract, and prescribes a component and its connectors within the architecture. Its attributes consist of Terrain and Pfmr_Renderer objects. In order to fully understand the Simulation class, the Pfmr_Renderer class must also be examined.

Just as ObjectSim prefaced the name of its Flt_Model class to indicate its dependence on a particular software package, the Pfmr_Renderer class is named to show its reliance on the Performer library. Originally this class was designed to collect all of a simulation's Performer dependencies in one module, so that ObjectSim would be more portable [Sny94]. While this concept is commendable, it was never realized, as all of the ObjectSim classes are dependent on Performer in varying degrees. Besides isolating dependencies, the Pfmr_Renderer class maintains lists of Players and Views, and effectively only provides the functionality for drawing the simulation. In other words, the Pfmr_Renderer class is functionally oriented, and the operations that it defines operate on attributes which should belong to the Simulation class. This anomaly is evident from the discussion of the design of the ObjectSim Simulation class, which explains that the Simulation performs state transitions depending on Pfmr_Renderer operations [Sny93,Fig14]. Finally, the Pfmr_Renderer class contains a Performer related attribute, *pipe*,which stores a representation of the processors on which Performer executes its application, cull, and draw threads (see Section 2.4).

The Easy_Sim software architecture defines an abstract *Simulation* class by combining ObjectSim's Pfmr_Renderer and Simulation classes. The elimi-

nation of Pfmr_Renderer preserves the integrity of the object-oriented data abstraction architectural style employed by Easy_Sim. Like its namesake in ObjectSim, the Simulation class in Easy_Sim serves as the glue that holds the other pieces of the application together. The operations of the Easy_Sim Simulation class are drawn from its two predecessors.

The ObjectSim Simulation class has one main operation, *propagate*. This operation is abstract, and must be overridden by a subclass to update each frame of the simulation. The Simulation class also provides two separate initialization operations, *alloc_shared* and *init_sim*. Operations in the Pfmr_Renderer class include *render*, which is the continuous loop that directs the entire simulation, *arbitrate*, which opens screen windows and establishes callback operations for the customization of Performer's cull and draw threads, and *insertmodel*, which adds an entity to the simulation. ObjectSim's Pfmr_Renderer and Simulation classes are represented by the Rumbaugh diagrams in Figure 22.

Easy_Sim's Simulation class has attributes for a Performer Pipe, an Environment, a Player_Manager, a Model_Manager, and a View_Manager, and it builds these data structures by means of three overloaded *Add* operations. The first of these operations adds an Environment and its Model into the Simulation for a given View. The second adds a View along with its Modifier, its attached Player and its offset from the Player, into the View_Manager. The final Add operation stores a Player with its Model and its Coords in the Player_Manager for a given View. Each Add which deals with a Model also assigns that Model using the Model_Manager.

In addition to the Add operations, the Simulation class also provides constructors, an *Open_Window* operation, a *Render* operation, and an abstract *Update* operation. These operations correspond to the alloc_shared and init_sim, arbitrate, render, and propagate operations in ObjectSim. Some of the

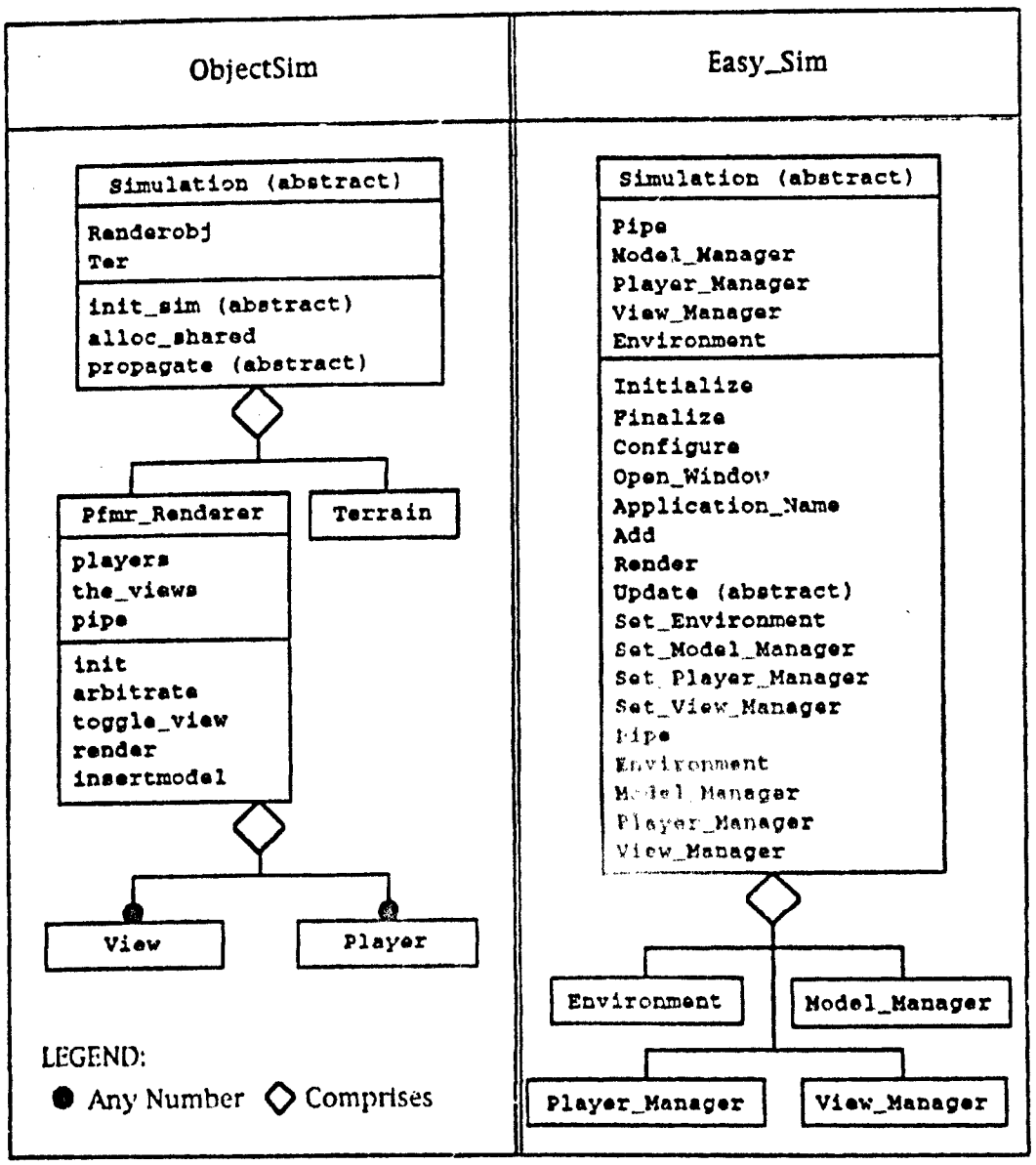


Figure 22. Simulation Class Object Model Diagrams

functionality assigned to the Pfmr_Renderer operations, such as multiple Player and View management, are moved to Easy_Sim's manager classes. Easy_Sim supplies an abstract *Application_Name* operation, so that application developers can customize the title of their simulation windows. Figure 22 shows a Rumbaugh diagram of the Easy_Sim Simulation class.

3.9 Summary of Easy_Sim Design

Figure 23 shows a Rumbaugh diagram outlining the relationships among the classes in the Easy_Sim architecture. To summarize the discussion in this chapter, the Model class stores database information describing how an object appears graphically, and this functionality is used by the Environment, which serves as the visual background for the application. The Model class may also represent Players, which are the entities whose interaction defines the application. A single Model may be cloned to represent multiple Players of the same type. The application is seen through Views, which must be attached to a Player, and may be changed by a Modifier. A Simulation ties the application together and encompasses the Environment along with manager classes, each of which administers multiple copies of their respective namesake classes. The diagram implies that the Simulation class has knowledge of the Model, Player, and View classes.

Figure 24 shows the language independent architectural model for Easy_Sim. Easy_Sim retains the architectural style of ObjectSim as a layered, heterogeneous system (see Section 2.2). The Easy_Sim layer follows the object-oriented data abstraction style, and defines the basis for components and their connectors in an application. Under the Easy_Sim layer sits the Performer layer, which is a case of the pipe and filter model. At the core of the Easy_Sim

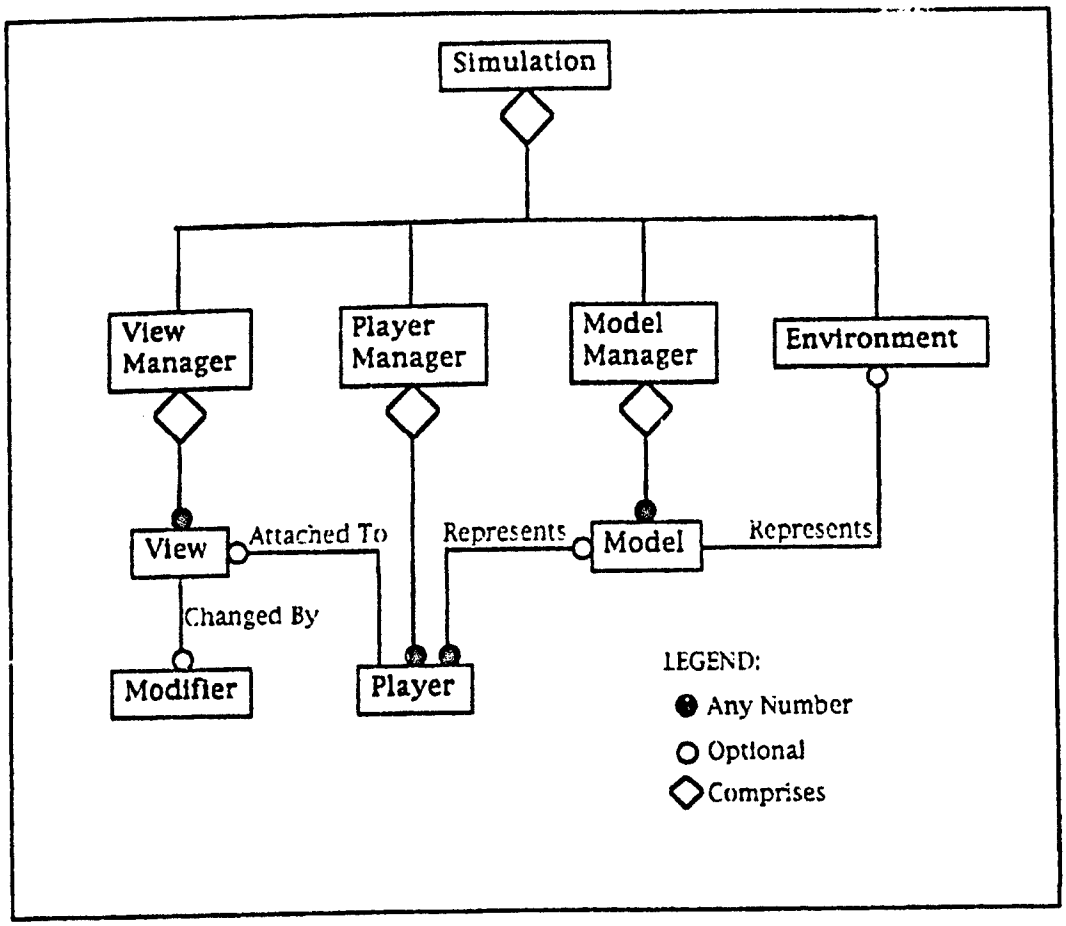


Figure 23. Simulation Class Object Model Diagrams

architecture lie the GL rendering library and the IRIX operating system, but they exist at a low level of abstraction and are generally omitted from discussion throughout this thesis. A developer builds an application on top of this structure, mainly interacting with Easy_Sim. Because the lower libraries provide powerful capabilities, the architecture retains the ability for the developer to access them directly.

Ideally, the Easy_Sim architecture should be portable, and its underlying layers should be interchangeable with a graphics library from any platform. There is nothing inherent in the Easy_Sim architectural design that

prevents this adaptation, but no industry standards for graphics libraries currently exist. The architectural connectors that allow interaction between the Easy_Sim and Performer layers therefore operate differently than connectors to other commercial graphics libraries would operate. Easy_Sim attempts to minimize the points where any modification would be necessary, but evaluation of these attempts cannot be quantified objectively without attempting to migrate an implementation.

This chapter has discussed the architectural design issues involved in migrating from ObjectSim to Easy_Sim. The next chapter examines the implementation issues involved in the transition. Ada 9X solutions are shown that realize the Easy_Sim application framework.

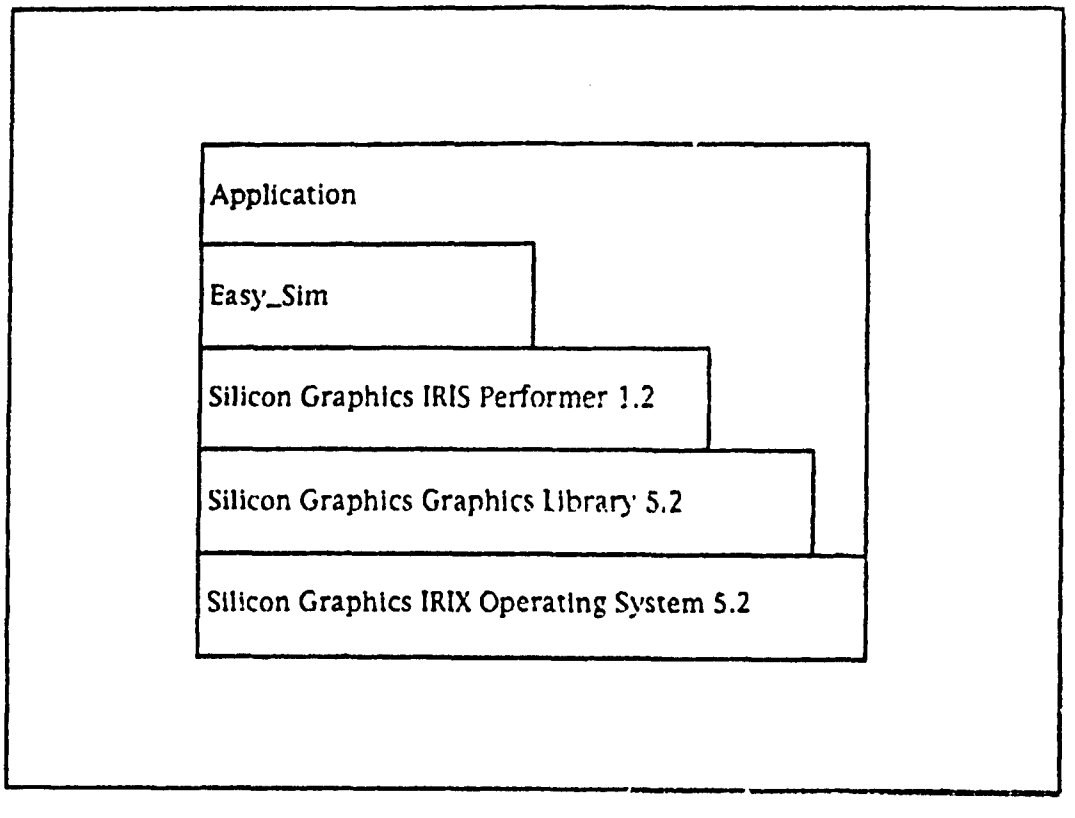


Figure 24. Easy_Sim Architectural Layering

IV Application Framework Implementation

This chapter examines the implementation phase of this thesis effort. It describes the transformation of the architectural design explained last chapter into the Easy_Sim application framework. This chapter describes the Ada 9X version of Easy_Sim, but a corresponding C++ version was also developed, so that a fair performance comparison could be made. The differences between these two versions are described in Section 6.3.

The discussion first covers general concepts that relate to the implementation effort. These topics include the ObjectSim framework, strategies for migrating ObjectSim to Easy_Sim, the framework's dependence on Performer, and the maturity of Ada 9X compilers. The chapter then presents the code template that Easy_Sim uses as the basis for each of its classes. The final portions of the chapter highlights the implementation issues specific to each Easy_Sim class. The details of the implementation can be found in the headers of the Easy_Sim code, which can be obtained by following the directions at the end of this document in Section 7.3.

4.1 General Issues

This section serves as a preface to discussing the implementation of Easy_Sim. It presents the underlying decisions that were made to form the general strategy for the production of the code. It first examines the ObjectSim application framework and describes the ideas considered for migrating this code to Ada 9X. This section also addresses Easy_Sim's dependencies on the Silicon Graphics Performer library and the GNAT compiler.

4.1.1 ObjectSim Implementation

Chapter III described the necessity model that Mark Snyder used to develop ObjectSim, and discussed how the code produced by the seven Graphics Lab students influenced ObjectSim's structure. While the necessity model allowed great gains in the Lab's productivity, it had negative effects on both the ObjectSim design and implementation. Snyder states in his thesis, "For each new problem solved, the major challenge was to fit it into the architecture while preserving a good design and not perturbing the existing code too much" [Sny93,72]. This section addresses the consequences of this approach.

Because the C++- language used to develop the ObjectSim framework was new to each of its contributors [Sny94], many of its positive features were not exploited. The encapsulation mechanism in C++ provides public, protected, and private sections in a class, and is used to prevent global manipulation of member data. ObjectSim makes free use of global access, however, as class members are generally public and modified by other classes. This lack of encapsulation results in ObjectSim's classes having low cohesion and high coupling, and causes logically abstruse code with untraceable effects. In addition to neglecting C++'s encapsulation mechanisms, ObjectSim also avoids constructors when initializing its classes. The omission of this basic C++ feature is rather curious, and results in the use of inconsistent initialization operations throughout the framework.

ObjectSim is further complicated by the contribution of code from seven individual developers. First, the differing styles, naming conventions, depth of comments, and coding idioms make various portions of the framework inconsistent and hamper its understandability. Second, the rapidity with which code contributions were integrated often results in a lack of proper documen-

tation. Portions of the ObjectSim code remain a complete mystery. The lack of encapsulation further exacerbates this matter, as an attempt to modify code in one class often causes unexpected behavioral changes in another class.

The next section describes how the negative aspects of the ObjectSim application framework affects the development of the Easy_Sim framework.

4.1.2 Easy_Sim Migration Strategies

Originally Easy_Sim was envisioned as an extension of ObjectSim, with the intent of exploiting reuse where possible. Because ObjectSim has proven such a success, its code was assumed to be a viable starting point for the production of Easy_Sim's code. However, the discovery of the problems described above caused contingency plans to be evaluated and executed.

Interaction with any foreign language from Ada presupposes a method of conversing with that language. This feat is accomplished by using a set of *bindings*. These Ada packages are filled with interface pragmas that tell an Ada compiler how to translate Ada entities to the foreign language. The Ada linker can then interact with the foreign object code and incorporate it into the Ada executable file. Two categories of bindings exist. A *thin* binding contains straight interfaces to the foreign language, and makes the caller translate complex parameters so that they are compatible with the foreign language's method for storing data. A *thick* binding is more robust and easier to use. Instead of simply providing interfaces, a thick binding provides subprograms which convert Ada parameter values to the correct format and then call the appropriate operation. All of the migration strategies for the Easy_Sim application framework use some form of bindings.

The first and perhaps most grandiose strategy considered for implementing Easy_Sim took reuse to the extreme. By convincing the compiler to derive Ada tagged types from C++ classes, Easy_Sim would act as a binding to the already proven ObjectSim framework. This idea was eliminated, however, because the design ObjectSim classes were deemed incompatible with the highly cohesive and encapsulated design envisioned for Easy_Sim. (Since the dismissal of this strategy, Thomas Quiggle of Silicon Graphics and Dr. Cyrille Comar of the GNAT Team have successfully demonstrated this technology [Qui94b]. It is unknown, however, if their solution can derive from C++ classes with the low cohesion of the ObjectSim classes.)

The second strategy considered for the implementation of Easy_Sim also reused the existing ObjectSim code. Although derivation from the abstract C++ classes had been discarded, reusing the concrete classes was examined. This strategy would build individual bindings to each of the member functions of each concrete C++ class, and would treat them as if they were regular C functions. This approach requires an additional step in designing the bindings, as the implicit C++ class pointer, *this*, must be passed explicitly in Ada [Qui94a]. This technique is further complicated if the C++ member function accesses variables outside its scope. Unfortunately, ObjectSim's reliance on global access caused the dismissal of this strategy.

The most practical strategy for the Easy_Sim development was also the most obvious strategy, and the Easy_Sim layer of the application framework is written entirely in Ada 9X. Although a mapping between an encapsulated C++ class and an Ada class package is straightforward, the low cohesion of the ObjectSim classes confounds the process. The final strategy adopted by Easy_Sim, therefore, implements its classes by using the ObjectSim classes solely as a ref-

erence base. Easy_Sim does not attempt to directly translate ObjectSim and omits some of the most complex functionality contained in its predecessor. Easy_Sim instead concentrates more effort on the architectural design and only provides the basic functionality needed to produce visual simulations.

4.1.3 Performer Dependencies

Although the final migration strategy chosen eliminates the need to interface with C++, it maintains interfaces to the C-based Silicon Graphics Performer and GL libraries. Performer manages all multiprocessing and drawing an application undertakes by efficiently processing complicated data structures every frame of the simulation. The efficient rendering of the models managed by Performer is accomplished by GL at a lower level. In order to reap the tremendous benefits offered by these libraries and achieve acceptable performance for realistic applications, Easy_Sim must rely upon both the Performer and GL libraries.

As both of these Silicon Graphics libraries are extensive, creating bindings to interact with them is a complicated undertaking. Luckily, developers at Silicon Graphics have built bindings to both libraries in order to produce their Paintball demonstration program [Emb94], and they have been gracious enough to contribute their work to this research effort. Because GL is older than Performer, its binding has evolved and become thick. The Performer binding is still in its infancy and is therefore thin. Easy_Sim makes many additions to the Performer binding, in fact, because the binding did not define interfaces to many of the Performer function calls used by Easy_Sim.

Completely freeing the Easy_Sim application framework from its Silicon Graphics based environment is necessary to make the application framework

fully portable, but this division is beyond the scope of this research effort. Additionally, in order to achieve a fair performance comparison with an ObjectSim implementation, an Easy_Sim implementation must use the same underlying architecture.

However, because portability is a long-range goal of Easy_Sim, Performer dependencies are isolated wherever possible. The first step in this process is to ensure that an application developed using Easy_Sim is not forced to access Performer. The second step is to push the Performer dependencies to the bodies of components, using Ada subunits where appropriate. This approach allows a new body to be written using a different graphics library, changing the implementation without affecting the interface and therefore saving expensive recompilations.

Easy_Sim has allowed its application developers to lessen their dependency on Performer and GL, and the example program described in Section 5.1 is testament to this independence. However, most developers realistically will want to access Performer and GL to benefit from their extensive capabilities. The second isolation step has been realized for GL, as Easy_Sim only calls its routines when opening the simulation window. This dependency is placed in a separate subunit, and the procedure can therefore be replaced easily.

While the vast majority of Easy_Sim's Performer dependencies have been placed in package bodies, some still remain as attribute and parameter types in package specifications. Theoretically, the Performer entities can be renamed or subtyped in one centralized location so that Easy_Sim's SGI reliance is obscure throughout the rest of the framework. Easy_Sim has proven this approach by subtyping Performer's coordinate types. This step has not

been taken for every Performer type, however, and it is left for future researchers.

Before analysis of the implementation strategy is described in full, the next section covers topics relating to the compilation system used to implement the Easy_Sim application framework.

4.1.4 Compiler Issues

Just as Easy_Sim's implementation in the SGI environment depends upon bindings to SGI libraries, it is also reliant on the existence of an Ada 9X compiler for SGI's IRIX 5.2 operating system. A team at New York University (NYU) has been sponsored by the Ada Joint Program Office to develop an Ada 9X compiler under GNU Public License for both Sun SPARC and IBM OS/2 systems. Their product is called the *GNU NYU Ada Translator (GNAT)*, and it is widely available on the Internet even though it is still under development. GNAT has been ported to run on many operating systems other than those for which it was originally targeted. Luckily, the developers at Silicon Graphics have ported GNAT to IRIX 5.2 for use with Paintball, and they have agreed again to contribute their work to this thesis effort.

GNAT is part of the Free Software Foundation's gcc compiler family. Gcc accepts code in a wide variety of languages, transforms that code with a language specific front end, and generates executables with a common back end. Use of this common back end makes GNAT quite mature for its age, as gcc has been continually improved over the last decade. However, the Ada specific front end is still rather immature and has plenty of room for growth.

A new version of GNAT is released roughly once a month. The version of Easy_Sim baselined for this thesis is compiled under GNAT version 1.83. Un-

fortunately, this version does not fully implement every Ada 9X features that Easy_Sim attempts to use, and alternate solutions are developed in these infrequent cases. Private extensions for derived tagged types is the most important of these incomplete features. To correct this problem, the type extensions used in most Easy_Sim classes remain public. Other compiler problems include the destructor *Finalize* not being called automatically, and occasional semantic confusion when a nested function call is misinterpreted. Neither of these cases cause much trepidation, thankfully, as Finalize is called explicitly, and the data obtained with function calls is accessed directly.

GNAT version 1.84 has been released on a small scale since the baselining of Easy_Sim, and the arrival of its successor, 2.00, is due. The GNAT development team has addressed the problems above, as they use Easy_Sim as one of their test cases. This thesis therefore assumes that the problems mentioned above have been rectified, and presents the implementation of Easy_Sim as if a complete Ada 9X compiler were available.

Discussion now concentrates on the Easy_Sim implementation, starting with the package that serves as the parent for the Easy_Sim framework.

4.2 The Easy_Sim Parent Package

The Easy_Sim application framework's implementation in Ada 9X employs hierarchical library units to create a large, logically related subsystem, while maintaining small, physically distinct pieces. This notion creates a cohesive and understandable hierarchy that remains modular and manageable.

At the root of this subsystem lies the Easy_Sim package. Nothing needs to be placed inside this package, but it must exist to serve as the skeleton that provides the basic structure for the rest of the hierarchy. For convenience,

entities commonly used throughout the Easy_Sim framework are placed within the parent package, because all declarations in the parent are visible to any child units. The coordinate types from Performer are subtyped here, because they are used by many different classes. An operation to read coordinates from a file, Read_Coords, is also localized with the type on which it operates. The context clauses for packages accessed throughout the framework are declared ahead of Easy_Sim. These clauses include the Performer bindings and Ada.Finalization, which declares the root types for controlled types.

Before analysis of each class in the Easy_Sim application framework occurs, the next section describes the commonalities that exist in each class.

4.3 Easy_Sim Class Package Conventions

Because the classes in Easy_Sim use the ROMAN-9X technique for their Ada 9X implementation, they each share common naming conventions, coding idioms, and a general structure (see Sections 2.1.4 and 3.2). This section presents the code that all classes have in common, so that is not repeated throughout the discussion. It also addresses the additions to ROMAN-9X that were added especially for Easy_Sim. .

Figure 25 contains the general outline of the code used to a class in the Easy_Sim application framework. A hierarchical library unit encapsulates each class, and this package is a child of the framework encompassing package, Easy_Sim. The class is defined as a controlled tagged type called Object, and may be abstract. New attributes used to extend this type are declared in the private part. Each class provides a classwide access type, and this type is called Reference. Controlled operations are declared, and a parameterized initialization operation, Configure, is also supplied. The parameters to Configure

are often given default values, so that they only need be addressed in unique cases. The class package then declares its main operations, which usually include Update and Draw, and they may be abstract or provide default behavior. The Set procedures and Get functions are then declared for the attributes for which they are needed. The Set procedures are named explicitly Set_(Attribute), while the Get functions are named by the attribute they return. Because the Get and Set subprograms are usually simple, *pragma Inline* is applied to them to optimize the code's execution time. In the actual type extension, attributes that are instances of another class are stored as Other_Class.Reference. This indirection is necessary so that the attribute can be passed to and from class operations, including the simple Get and Set operations, without violating the one tagged subtype per operation rule [RM94, 3.9.2.12]. The use of a pointer also more closely models a real world situation, where only one copy of each entity exists.

Two categories of Easy_Sim classes are declared at another level of depth within the Easy_Sim framework. The manager classes exist as hierarchical library units within the class they manage. The Player_Manager class package, for example, is declared as Easy_Sim.Player.Manager, logically adding a Manager child package to the Easy_Sim.Player package. This technique of embedding packages is also used for the default implementations of the classes such as Easy_Sim.Environment.Terrain and Easy_Sim.Modifier.Standard_Input that are provided for applications developers.

```
with Easy_Sim.Other_Class;

package Easy_Sim.Class is

   type Object is abstract new Ada.Finalization.Controlled with
      private;

   type Reference is access all Object'Class;

   procedure Initialize (Instance  : in out Object);
   procedure Adjust     (Instance  : in out Object);
   procedure Finalize   (Instance  : in out Object);

   procedure Configure  (Instance  : in out Object;
                         Parameter : in     Parameter_Type := 0);

   procedure Update     (Instance  : in out Object) is abstract;
   procedure Draw       (Instance  : in out Object) is abstract;
   procedure Operation  (Instance  : in out Object;
                         Parameter : in     Parameter_Type);

   procedure Set_Attribute_A
      (Instance       : in out Object;
       To_Attribute_A : in     Attribute_Type);
   procedure Set_Attribute_B
      (Instance       : in out Object;
       To_Attribute_B : in     Easy_Sim.Other_Class.Reference);
   pragma Inline (Set_Attribute_A, Set_Attribute_B);

   function Attribute_A (Instance : Object) return Attribute_Type;
   function Attribute_B (Instance : Object)
      return Easy_Sim.Other_Class.Reference;
   pragma Inline (Attribute_A, Attribute_B);

private

   type Object is abstract new Ada.Finalization.Controlled with
      record
         Attribute_A : Attribute_Type := Attribute_Initial_Value;
         Attribute_B : Easy_Sim.Other_Class.Reference;
      end record;

end Easy_Sim.Class;
```

Figure 25. General Easy_Sim Class Format

The package specification in Figure 25 represents the basis for all of the class packages corresponding to the Easy_Sim architectural components. The rest of the chapter describes the features of each class that make its

implementation unique. Discussion starts at the bottom of the dependency chain, and progresses until the Simulation is reached. The code to accompany this discussion is not included in the text due to its length, and the figures in Chapter III can help the reader follow discussion. Section 7.3 describes methods by which the reader can obtain a copy of the code.

4.4 The Model Classes

This section discusses the Model and Model_Manager classes, which are remarkable because of their simplicity. The power of Performer accounts for this positive property, as the library supplies most of the functionality. Model's Configure operation, for instance, turns graphical database geometry built from many different tools into a Performer rendering tree node with one call to Performer's *LoadFile* function. The remainder of this section examines the Adjust operation in the Model class and briefly describes the workings of the Model_Manager class. Figure 16 shows the Rumbaugh diagram of the Model class, while the Model_Manager can be found in Figure 21.

The Model class is unique among the Easy_Sim classes because it is the only class whose Ada type, *Object*, is controlled, instead of limited controlled (see Section 2.1.2). This trait follows from the need to copy instances of the class (see Section 3.3), which does not exist elsewhere. The Model class provides both a procedure and function to *Clone* Models, and both of these rely on the controlled Adjust procedure and a Performer Clone function to accomplish their objective. The Adjust procedure, however, was not as straightforward to use as originally envisioned. Because it only takes one Model.Object parameter, the notion of source and target for assignment inside Adjust becomes rather clouded--the parameter must serve as both. The solution is to start with

a temporary variable as the target and the parameter as the source. Performer's Clone function is called on the parameter, with the result saved to the temporary variable. The final step is to assign the temporary back to the parameter. This step must be accomplished using the record components of the Model.Object, as assigning the Model.Object itself will result in infinite recursion. The remainder of the Model class is implemented in a straightforward manner.

The container that differentiates Models, the Model.Manager class, uses the file name containing a database to distinguish various Models. It stores a list of these file names and their accompanying Models. The Assign_Model procedure is given a string parameter, *File_Name*, which it compares against the names in the list using the private *Index* function. If a match is found, the Model being assigned is copied from the Model at the appropriate index in the list, using the Model.Adjust procedure. If no match is found, Model.Configure is called to convert File_Name into a rendering tree, and the new Model and File_Name are stored in the list. The reader should note that if two distinct files contain the same Model, the Model.Manager cannot recognize this equality. The developer can also use this trait if, for some reason, he does not wish Model cloning to occur.

The next section examines the interesting aspects of the Environment class and its descendants.

4.5 The Environment Classes

This section analyzes the Environment classes in the Easy_Sim application framework. It begins with the abstract base class, then discussion turns to the Horizon and Sun building blocks, and the section concludes by describing the

default Terrain class. Figure 17 shows an Object Model diagram of the Environment classes.

The abstract Environment class works quite simply. The Initialize operation allocates the Model that represents the simulation's background. The Set_Model procedure is used to assign the Model, and the Image function returns the root of the Model's rendering tree. The Configure procedure places the Model's local tree into the Simulation's tree by using a Performer call to add the Model to the tree node which it is given as a parameter. This technique is used throughout the Easy_Sim classes, in fact, wherever a Model needs to be placed in the Simulation's rendering tree. The only other operation in Environment is the abstract Draw procedure, which forces a subclass to declare if it intends to do unique rendering on the draw thread. Examples of this special drawing includes landscape grids or text overlays.

The Terrain class is derived from Environment and joins the basic functionality of the Horizon and Sun classes to model the earth, sky, and sunlight. All of these classes were based on the ObjectSim Simple_Terrain class, but modularizing the previously monolithic class has allowed for greater possibilities of future reuse at both the design and coding levels.

The Horizon class uses the basic Performer earth/sky capabilities, *ESky*. It establishes a green earth and a blue sky that gets lighter with increasing altitude. Its values are hard coded and taken from many of the examples provided with Performer. Performer requires the Esky to be attached to a window, or Performer Channel, as it does many items. The Configure operation awaits this parameter and makes the proper call to fasten the Esky to the window. Performer automatically draws the Horizon for the rest of the simulation, and no Update procedure is necessary for the class.

The Sun works similarly, but has attributes for its *Position* and *Ambience*, and is an instance of the Performer *LightSource* type. Ambience is similar in effect to brightness. In the Initialize constructor, the LightSource is allocated, the Position and Ambience are given default values, and the Sun is set to shine from directly overhead. Performer requires a LightSource to be included in the rendering tree, and the Configure operation places the Sun in the tree as Environment.Configure did above. Once the Sun is part of the rendering tree, Performer automatically draws it every frame and no Update procedure needs to be provided.

The Terrain class brings together the Environment, Horizon, and Sun in a direct manner that shows the benefits of the building block approach. The Terrain class has both Horizon.Reference and Sun.Reference attributes. Its Initialize calls its parent's Initialize and invokes its attributes' Initializes by allocating them. Terrain.Configure likewise calls the Configures for its parent and components. Terrain must override the abstract Draw procedure it inherits from Environment, and it makes this procedure null. Because Performer handles the execution of its attributes, it also transitively Updates the Terrain, and after initialization is complete, there is nothing more to do.

This section has described the ease with which a default Environment can be created due to the building block approach. The next section highlights the implementation of the Player and Player.Manager classes.

4.6 The Player Classes

This section examines the interesting aspects of the Player and Player_Manager classes of the Easy_Sim application framework. The Rum-

baugh Object Model diagram for the Player class is found in Figure 18, and Figure 21 shows the Player_Manager class.

The abstract Player class provides the basis for entities within the simulation. Its controlled Initialize procedure allocates its Model attribute, and its Configure procedure adds the Model's geometry under the given node in the rendering tree. Both the Move_Straight and Look_At procedures use analytic geometry principles to calculate the new Position and Direction of the Player. An instantiation of Ada 9X's new Ada.Numerics.Generic_Elementary_Functions package is used in Move_Straight to calculate the Sin and Cos.

The Player class provides an extensive set of Get and Set operations so the client programmer can gain access to encapsulated data. The Model attribute is accessed by both Image and Model Get functions, with the Image returning the rendering subtree for that Model. The Coords attribute, which contains two arrays of three values each, has flexible Get and Set components which return world coordinate values to a client programmer. These operations include *Position*, *X_Position*, *Y_Position*, *Z_Position*, *Direction*, *Heading*, *Pitch*, and *Roll*, in addition to Coords. This final operation is overloaded and can be addressed as a whole or as a Position and Direction pair. Because the Coords operation cannot return two values, the Get operation for the second Coords is a procedure called *Get_Coords*. Finally, the Player class provides an abstract Update procedure through which a client programmer must define the behavior of the subclass.

The Player_Manager class provides default administration of the Players who interact in the Simulation. It stores, *List*, an array of the Players as well as the *Count* of Players in List. The Update procedure simply calls Update for each of the Players in List. The Add procedure takes in a Player and a node

under which the Player gets stored in the rendering tree. Add increments Count, adds the Player to List, and calls Player.Configure passing the tree node under which the Player's geometry will be attached. Client programmers can override the simplistic operation of the Player_Manager if they so choose.

This section has highlighted the implementation of the Player and Player_Manager classes. The next section looks at the abstract Modifier class.

4.7 The Modifier Class

The abstract Modifier class is intended to provide the means through which a user of an Easy_Sim application can change the View. Because the Modifier does not have access to the View, however, it simply stores its own state and lets the View read that state to update itself. This section looks at both the abstract base class and the Standard_Input default subclass. Figure 20 shows the Easy_Sim Modifier class hierarchy.

The abstract Modifier class contains a Coords attribute that maintains the offset between the View and the position and direction to which the user has moved and pivoted. The Modifier therefore has a slew of Get and Set attributes to access these values. Unlike the Player's global coordinates, the Modifier class maintains local coordinates, in that its Coords are an offset and pivot relative to the View. The Reset procedure sets the values of the offset and pivot all back to zero. The abstract Update procedure is the connector through which subclasses define how the Coords are changed.

The Standard_Input subclass of Modifier provides input values from the mouse and keyboard, and is only partly implemented. The package maintains a list of Boolean flags, each of which represents a key press or mouse click during a frame. The flags are maintained globally in the package specifica-

tion so that client packages can reset them after their use. A function, *Flag_Copy*, returns a pointer to the structure, and it is the preferred method through which the flags are accessed.

The Standard_Input class operates by means of Performer utilities that collect input from the mouse and keyboard through the simulation window. The Initialize constructor initializes both the Mouse and Keyboard components, and the Update procedure defers to *Read_Mouse* and *Read_Keyboard*, each of which checks the queues for each part of each device and updates the corresponding input flag if necessary. The Standard_Input package currently reads data, but it does not modify its inherited Coords attribute. This step was the next addition scheduled for the Easy_Sim framework when it was baselined for this thesis.

The Modifier class currently is the location in the Easy_Sim architecture where it is most sensible to store the input values that are used throughout the Simulation for executive control. This approach may not be the best model to achieve this effect. Section 7.2 contains discussion of alternate solutions.

This section has outlined the operation of the Easy_Sim Modifier class. The next section looks at the Easy_Sim View and View_Manager classes.

4.8 The View Classes

The View class provides the ability for the user to look into the Simulation, and each View can be thought of as a different window into the scene. A View must be attached to a Player within the Simulation. The View_Manager class is responsible for administering multiple Views, and it keeps track of the list of View states. Each View state contains a Player and the offset and rotation of

the last View that was attached to that Player. This section examines the implementation of the View and View Manager classes. Figure 19 shows the Rumbaugh diagram of the View class, while the View_Manager can be seen in Figure 21.

The View class maintains attributes for the Modifier and Player with which it is associated, its local Coords relative to the Player, and the Performer types Channel and Scene. The Get and Set attributes for the component classes are straightforward. The Performer types are controlled by the View, and Set procedures are not available for them, although they can be accessed with Get functions. The Set procedures for Coords all expect local coordinates, and the Gets on the entire Coords structure also return values relative to the Player. However, the Get functions all return world coordinate values, as they combine the local Coords with the Player's world coordinates by using matrix transformations.

The View's controlled Initialize procedure creates a new Scene that serves as the root of the rendering tree for the items in that View. The Configure operation is rather busy in the View class. The Configure takes the processing Pipe on which the Simulation is operating as a parameter, and the Pipe is required by Performer to allocate the Channel. Configure also fastens the Scene to the Channel and sets up default values for the angles that the View can see. Finally, Configure establishes the calibacks for Performer's cull and draw processes.

The Cull and Draw procedures provide default implementations for their respective threads for the given Channel. Cull calls Performer's cull function, while Draw first clears the Channel and then calls the Performer draw function. Pre_Cull, Post_Cull, Pre_Draw, and Post_Draw procedures are provided by

Easy_Sim, although they were omitted from Figure 19 due to space constraints. They are called before and after their respective Performer functions to allow maximum flexibility in application customization. They all default to null except Post_Draw, which makes the Performer utility call to collect user input.

Because of the currently unresolved incompatibilities between the C-based Performer execution and the Ada 9X callbacks, no Ada variables can be accessed inside any of the callback operations, either globally or through the parameter list. However, as this process was not necessary to achieve a working simulation, not much effort was put forth in finding a solution. Section 7.2.2 addresses possible corrections to this dilemma.

The incompatibility between the languages also affects the ability of the application developer to customize the callback operations. Because the Configure operation establishes the procedure to which the callback occurs, it must be overridden first. The new Cull or Draw procedure must also be overridden, and then any Pre or Post operations on it can also be redefined. While this process seems like extra work, C++ also does not allow callbacks to virtual functions, and a similar strategy must be applied there.

The final View class operation, Update, first updates its local Coords with any new changes from the Modifier. It then calls its own Get functions to obtain its Position and Direction in world coordinates, and it passes this information to Performer so that the View can be placed correctly to look into the Scene.

The View Manager class maintains two lists, the multiple *Views* and the View *States*. The controlled operations and Configure currently are all null operations. Because no testing has been performed with multiple Views, their discussion here is limited. The remainder of this section discusses View States.

Multiple Views can be added to the Views list by the Add procedure. Like the Player_Manager.Add, this procedure increments the Count, places the new View into the Views list, and calls Configure on the new View passing the Pipe parameter it was given. The Set_View procedure takes a given View, finds it in the Views list using an *Index* function, and activates it by setting a Boolean flag. The Update procedure iterates through the Views list, and calls Update on the Views which are activated.

The Set_Player procedure takes a View and a Player with the intent of attaching the View to the Player. The use of View States allows the View to be placed at the same offset and rotation from the Player as the last View that was attached. In order to achieve this effect, the first step in this procedure is to remember the current state of the Player being detached. This step is accomplished by a procedure within the body of the package, *Save_State*. The next step is to call the View.Set_Player procedure. Finally, the Coords of the View must be assigned to the last state associated with the new attached Player. This step is performed by another hidden subprogram, the *Offset* function, which recovers the old state from the States list.

This section has summarized the implementation of the View and View_Manager classes. The final section of this chapter highlights the operation of the Easy_Sim Simulation class, which is the capstone of the Easy_Sim framework.

4.9 The Simulation Class

The abstract Simulation class provides the structure to bring all of the pieces in a visual simulation together. The basic interaction with Performer occurs in this class, establishing the foundation of the application. This section de-

scribes the implementation of the Simulation class. Figure 22 shows the Rumbaugh Object Model representing the Easy_Sim Simulation class.

The controlled operations for the Simulation class respectively call the *Init* and *Exit* functions of Performer. The Configure procedure has a long list of parameters, all of which provide information to configure Performer differently, and all of which have normal default values. These parameters address such items as the *Frame_Rate* desired for the application, the *Input_Mode* that will be used to gather user input, the *Message_Level* describing the amount of processing information the user would like displayed to the console, and so on. Configure uses all of these parameters to prepare the application, and finally calls Performer's *InitPipe* function which performs a callback to Open_Window to start the Simulation.

Open_Window is implemented as a separate subunit because it is the only part of Easy_Sim which uses GL functions. It calls Application_Name as it issues the window opening command so that the application developer can customize the title of the window in which the simulation is rendered. Just as the callbacks in the View class have inter-language communication problems, so does the Open_Window procedure. Any attempt to access the application name globally or through parameters fails, and the function call is used because it works.

The Simulation class has five attributes. Environment, Model_Manager, Player_Manager, and View_Manager are all class References, and they have both Get and Set attributes. The Set attributes for the manager classes act slightly differently than normal Set operations, because they all take null values by default and allocate themselves the first time they are called. This implementation was chosen because of the privacy problem caused by subclasses

not being able to access the components of their parent (see Section 6.3). Alternatively, the managers could all be allocated in Initialize, but this approach prevents them from being overridden easily. The fifth attribute is Pipe, the processor model from Performer. This value is initialized in Configure and cannot be changed. Pipe therefore only has a Get function associated with it.

The Simulation class declares three overloaded Add procedures, to respectively add the Environment, a Player, or a View to the Simulation. These operations are where the Simulation ties all of its pieces together. The Add View must be called before the others, so that they can be placed in a View. Add View takes a *New_View*, a Player to which it will be attached, a Modifier if one is needed, and either a set of Coords or a file from which the Coords can be read. The procedure first calls View.Manager.Set_Player with the given Player and then calls Set_Modifier. These steps fasten these items to the View. The next step calls View.Set_Coords to set the offset from the Player, either by assigning the given Coords, or by calling Read_Coords to retrieve the data from a file. The latter method allows applications to be easily changed without recompilation. Finally, the Add View procedure calls View.Manager.Add, and the View becomes part of the Simulation.

The Add Environment procedure takes parameters for the *New_Environment*, a View to which is attached, and a *Model_File* containing the graphics database. The procedure first calls Model.Manager.Assign_Model and Environment.Set_Model to get the Model and attach it to the Environment. Environment.Configure is called next, passing the Channel and rendering subtree so that any necessary linking between the Environment and the View can occur. The final step is to call Set_Environment with the

New_Environment, ensuring that the Simulation is also properly tied to its background.

The third and final Add procedure adds a *New_Player* to the Simulation, and it combines the features of the other two Adds. Add Player takes parameters for the View it appears in, its *Model_File*, and either method of passing its Coords. Like Add Environment, it first calls Model.Manager.Assign_Model and Player.Set_Model. Like Add View, it then calls Player.Set_Coords, perhaps by reading from a file first. Finally, Add Player calls Player.Manager.Add to ensure that the Player becomes part of the Simulation.

The culmination of the Simulation occurs in its Render procedure. This procedure loops continually, causing Performer to draw the application on the display. Render first calls Performer's *Sync* function to coordinate all three Performer threads on a frame boundary. It then calls View.Manager.Update to Update the positions of all the Views. Render's call to the Performer *Frame* function causes the start of the cull and draw threads. The Player.Manager.-Update is called next, to move all the Players in the Simulation. Finally, the Simulation calls its own abstract Update procedure, allowing any overriding functionality of its subclass to be incorporated into the main rendering loop. Usually, user input that controls the flow of the application is processed here, possibly ending the loop. Otherwise, another frame is drawn, and so on...

This chapter has described the implementation of the Easy_Sim application framework. Most of the discussion here holds in both the Ada 9X and C++ versions, with the differences described in Section 6.3. Instructions on obtaining the Easy_Sim code appear in Section 7.3. The next chapter describes an Ada 9X implementation of an example application using the Easy_Sim software architecture and application framework.

V Example Application

This chapter describes how to develop a basic visual simulation using the Easy_Sim application framework. It first does this by presenting an example application, the Circling Planes. The example leads the reader through the entire application development process. The chapter ends by illustrating general guidelines for deriving from the different Easy_Sim classes to develop any application.

5.1 The Circling Planes Example Application

This section covers the complete development of an Ada 9X solution for an application. The overall plan for the simulation is described first, and then a detailed look at each of the necessary components follows.

This simulation, *Circling Planes*, involves two aircraft circling over a ground-based background and a observation point from which the planes can be tracked. The simulation's view can be attached to either of the planes or the tracker point, and can be switched between these entities by using the mouse. This application is taken from Example 2 in the ObjectSim Application Developer's Manual [Sny93,A7-16].

The Circling Planes simulation mainly uses the default classes supplied by the Easy_Sim framework, but it also derives application specific components from Easy_Sim's abstract classes. Figure 26 shows an Object Model diagram representing the overall design of the system, with the Easy_Sim framework shown in the top portion, and the application specific classes shown in the lower portion. Bold lines show inheritance relationships, while normal lines show aggregations and regular associations.

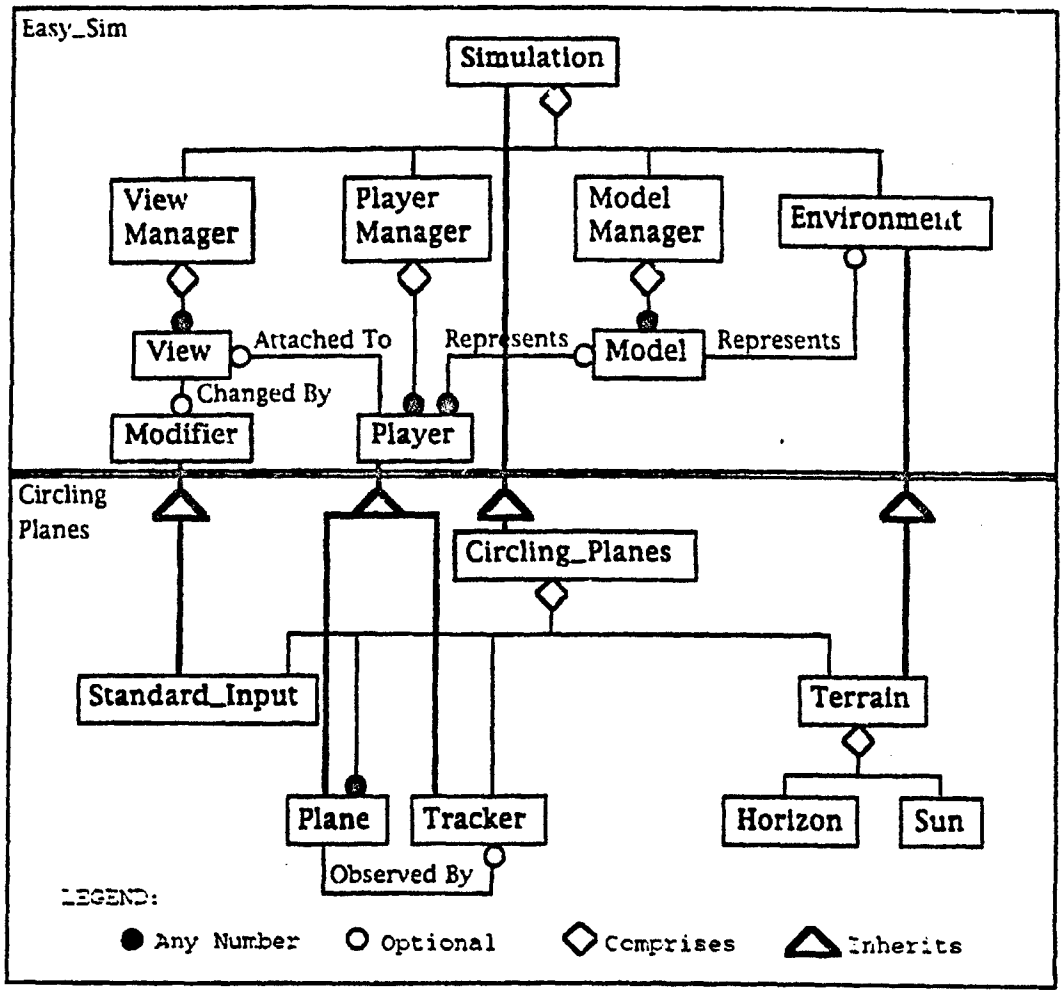


Figure 26. Circling Planes Simulation Object Model Diagram

The Player class provides the basis for both the Plane and Tracker classes, and each Tracker is related to the Player it follows. (This association is restricted to Planes in Figure 26 to keep too many lines from crossing.) The application background uses Easy_Sim's standard Terrain class, and therefore includes basic representations for the Horizon and Sun. Input is entered into the application through the use of Easy_Sim's Standard_Input Modifier class,

which provides services for keyboard and mouse entries. Finally, the Circling_Planes class is derived from Easy_Sim's Simulation, and serves to connect all of the pieces to make the application.

The remaining portions of this section concentrate on each of the classes derived to make the Circling Planes example. Because these classes are not part of the Easy_Sim application framework, they are not declared as child packages of Easy_Sim. Instead, they each stand alone. The Plane class is the first to be examined.

5.1.1 The Plane Class

In order to simulate an entity within an Easy_Sim application, a subclass of the Player class must be defined. To simulate a plane, the new class is appropriately named *Plane*. An instance of the Plane class moves by simply moving forward and changing its heading to the right each frame, making a clockwise circle.

To implement this design in Ada following the Easy_Sim architecture, a class package named Plane is created which derives its class type Object from Easy_Sim.Player.Object. This step requires access to Easy_Sim.Player through a *with* context clause. The Plane class does not need to add any attributes to the Player class, and inherits the majority of its parent's operations without modification. To be consistent with Easy_Sim structure, the Plane class declares a classwide access type, Reference, and places its attribute extensions in the private part. The circling movement of the Plane is accomplished by overriding the Player's Update procedure. Finally, the Player's abstract Draw procedure is also overridden, but because no special drawing is needed in this application,

```
with Easy_Sim.Player;

package Plane is

   type Object is new Easy_Sim.Player.Object with private;

   type Reference is access all Object'Class;

   procedure Update (Instance : in out Object);
   procedure Draw   (Instance : in out Object);

private

   type Object is new Easy_Sim.Player.Object with null record;

end Plane;
```

Figure 27. Plane Class Package Specification

this operation does nothing. Figure 27 contains the Ada package specification for the Plane class.

The body of the Plane package is slightly more complex. The body first shows its dependence on the Performer's pf library by accessing it via the *with* clause. The Plane implementation also introduces the Basic_Types package, which is implemented to provide Ada types analogous to the C types used in Performer. The Update procedure then moves the Plane forward by calling its inherited Move_Straight procedure, and decrements the Plane's heading by using Get and Set operations and a temporary variable. The use of this variable ensures that the Plane's heading stays within a realistic range. The final step in Update is to change the coordinates for the Plane in the rendering tree, and this step is accomplished by means of a Performer call. The implementation of the Plane class is completed by providing a dummy procedure to override the Draw procedure. Figure 28 shows the body of the Plane package.

```
with Performer_Pf;
with Basic_Types;

package body Plane is


   procedure Update (Instance : in out Object) is

      New_Heading : Basic_Types.Float32 := 0.0;

   begin

      Move_Straight (Instance, 4.0);

      New_Heading := Heading (Instance) - 0.5;
      if New_Heading < 0.0 then    --  Wrap around if full circle
         New_Heading := 360.0;
      end if;
      Set_Heading (Instance, New_Heading);

      Performer_Pf.PfDCSCoord (Image (Instance), Coords (Instance));

   end Update;


   procedure Draw (Instance : in out Object) is
   begin
      null;
   end Draw;


end Plane;
```

Figure 28. Plane Class Package Body

5.1.2 The Tracker Class

To observe the circling planes, another Player subclass is defined, the *Tracker*. This class has no model of its own, and merely exists to watch another Player. Like the Plane class, the Tracker class inherits the majority of its attributes and operations from Player. It does add an attribute however, *Trackee*, to store a Reference to the Player being tracked. It also adds a Set operation to modify its new attribute. Tracker overrides Update to define its tracking techniques, and must override Draw to become concrete. Tracker also redefines Initialize

```
with Easy_Sim.Player;

package Tracker is

   type Object is new Easy_Sim.Player.Object with private;

   type Reference is access all Object'Class;


   procedure Initialize (Instance : in out Object);

   procedure Update     (Instance : in out Object);

   procedure Set_Trackee
      (Instance   : in out Object;
       To_Trackee : in     Easy_Sim.Player.Reference);
   pragma Inline (Set_Trackee);


private

   type Object is new Easy_Sim.Player.Object with
      record
         Trackee : Easy_Sim.Player.Reference;
      end record;

end Tracker;
```

Figure 29. Tracker Class Package Specification

here to demonstrate this process, even though it is not necessary. Figure 29 outlines the Tracker package specification.

The body of Tracker is similar to Plane, but due to its lack of a Model, has no dependency on Performer. The Initialize procedure here is superfluous, but serves to show the concept of *view conversion*. To reuse the steps in initializing its inherited components, Tracker.Initialize simply calls its parent's operation. Logically, this call passes only the components of the parent record type, and Tracker is free to initialize its new components afterwards. The Update procedure uses the inherited Look_At procedure to orient the Tracker towards the position of the Trackee. It finds this position by dereferencing its component's pointer and calling the Get operation on the Player.Object'Class to find its location. This technique works on any Player subclass, as the tag of

```
package body Tracker is

   procedure Initialize (Instance : in out Object) is
   begin
      Easy_Sim.Player.Initialize (Easy_Sim.Player.Object (Instance));
      Instance.Trackee := null;
   end Initialize;


   procedure Update (Instance : in out Object) is
   begin
      Look_At (Instance, Easy_Sim.Player.Position
                            (Instance.Trackee.all));
   end Update;


   procedure Set_Trackee
      (Instance    : in out Object;
       To_Trackee : in     Easy_Sim.Player.Reference) is
   begin
      Instance.Trackee := To_Trackee;
   end Set_Trackee;


   procedure Draw (Instance : in out Object) is
   begin
      null;
   end Draw;

end Tracker;
```

Figure 30. Tracker Class Package Body

Trackee's designated object dispatches the call to the appropriate Position function. Figure 30 contains the body of the Tracker class package.

5.1.3 The Circling_Planes Class

As stated in the design chapter, the Simulation class serves as the organizational fulcrum of the application, tying together all of the simulation's pieces. This application, *Circling_Planes*, is no different. The context clauses show that it uses the Easy_Sim classes as building blocks. The class type is derived from Easy_Sim.Simulation.Object, and its attributes define the entities involved

```
with Easy_Sim.Player;
with Easy_Sim.Modifier;
with Easy_Sim.View;
with Easy_Sim.Simulation;

package Circling_Planes is

   type Object is new Easy_Sim.Simulation.Object with
      private;

   type Reference is access all Object'Class;

   procedure Initialize (Instance : in out Object);

   procedure Update     (Instance : in out Object);

   Quit_Program : exception;

private

   function Application_Name (Instance : access Object)
      return String;

   type Object is new Easy_Sim.Simulation.Object with
      record
         Main_View   : Easy_Sim.View.Reference;
         Tracker     : Easy_Sim.Player.Reference;
         Flight_Lead : Easy_Sim.Player.Reference;
         Wing_Man    : Easy_Sim.Player.Reference;
         Input       : Easy_Sim.Modifier.Reference;
      end record;

end Circling_Planes;
```

Figure 31. Circling_Planes Class Package Body

in the application. There is one View, *Main_View*; three Players, *Tracker*, *Flight_Lead* and *Wing_Man*; and one input device, *Input*. The constructor, Initialize, defines the relationships between the different attributes, and Update defines how input affects the Simulation each frame. An exception, *Quit_Program*, provides an avenue for gracefully exiting from the inherited Render loop. Finally an overriding private function, Application_Name, allows the window to be labeled with the title of this example. Figure 31 holds the code for the Circling_Planes specification.

```
with Performer_Pf;
with Performer_Pfutil;
with Easy_Sim.Environment.Terrain;
with Easy_Sim.Modifier.Standard_Input;
with Plane;
with Tracker;

package body Circling_Planes is

   procedure Initialize (Instance : in out Object) is (in Figure 33);
   procedure Update     (Instance : in out Object) is (in Figure 34);

   function Application_Name (Instance : access Object)
      return String is
   begin
      return ("Easy_Sim Circling Planes Example");
   end Application_Name;

end Circling_Planes;
```

Figure 32. Skeleton of Circling_Planes Class Package Body

The body of the Circling_Planes package establishes the relationships between the different entities in the simulation, and assigns the appropriate subclasses for the different attributes. These subclasses are accessed by the collection of *with* clauses that precedes the program unit. Figure 32 shows the Circling_Planes class body. The code is too large to show at once, however, so the figure only presents part of the body, and discussion of Initialize and Update follows below. The one operation shown, Application_Name, is called just before the window for the simulation is opened, and it provides the name that appears in the corner of the window. Because this function is specific to this application and only needs to be called once, it is declared in the private part of the package specification.

Figure 33 shows the Initialize procedure of Circling_Planes by itself. It first calls its parent's Initialize and Configure operations so that Performer and the Easy_Sim framework are initialized. Most of the parameters of Configure take their default values, but the Modifier in this application uses Per-

```
procedure Initialize (Instance : in out Object) is

   --  Initialization of Terrain must occur after PfConfig,
   --     which is called in Sim.Init, so allocation must wait
   Temp_Terrain : Easy_Sim.Environment.Reference;

begin

   Easy_Sim.Simulation.Initialize
     (Easy_Simulation.Object (Instance));

   Configure (Instance,
              Input_Mode => Performer_Pfutil.PFUINPUT_GL);

   Instance.Main_View   := new Easy_Sim.View.Object;
   Instance.Tracker     := new Tracker.Object;
   Instance.Flight_Lead := new Plane.Object;
   Instance.Wing_Man    := new Plane.Object;
   Instance.Input := new Easy_Sim.Modifier.Standard_Input.Object;
   Temp_Terrain   := new Easy_Sim.Environment.Terrain.Object;

   Add (Instance, New_View      => Instance.Main_View,
                  With_Player   => Instance.Wing_Man,
                  With_Modifier => Instance.Input
                  Coords_File   => "view.data")

   Add (Instance, New_Player  => Instance.Tracker,
                  Coords_File => "tracker.data");
   Tracker.Set_Trackee (Tracker.Reference (Instance.Tracker).all,
                        Instance.Flight_Lead);

   Add (Instance, New_Player  => Instance.Flight_Lead,
                  Coords_File => "lead.data",
                  Model_File  => "lead.flt");

   Add (Instance, New_Player  => Instance.Wing_Man,
                  Coords_File => "wing.data",
                  Model_File  => "wing.flt");

   Add (Instance, New_Environment => Temp_Terrain,
                  Model_File      => "terrain.flt");

end Initialize;
```

Figure 33. Circling_Planes Class Initialize Procedure

former's GL mode of reading input, and the default X Windows input mode is therefore overridden.

Initialize then allocates each of its attributes, and they are allocated with particular subclasses. Declaring the attributes in the specification as

base classes and allocating them in the body as subclasses allows more flexibility in their handling in the application. Attributes inherited from the Simulation class are private, so the Environment and various Manager attributes cannot be accessed or allocated directly here. The use of Temp_Terrain shows a method for bypassing this annoyance when necessary. The default manager classes all allocate themselves when they first are referenced to circumvent this privacy problem.

The relationships among the Environment, Modifier, Views, Players, and Managers are all incorporated into the simulation by using the various Add procedure calls and Set operations where appropriate. The Add operations allow files containing rendering models to be specified for the Environment and any Players that need them, and the Adds also allow the initial coordinates for the Views and Players either to be passed explicitly or read from a file. Using files for reading Models and Coords allows the developer to customize the application by simply changing the models or coordinates contained in a particular file. This flexibility saves inefficient recompilations should different values for these attributes need to be tested or used in a simulation.

The Update procedure of the Circling_Planes package, shown alone in Figure 3.1 allows user input to change the state of the simulation each frame. The user can press the capital *S* to toggle Performer's statistics displays or press the Escape key to quit the program. By using the mouse, the user can attach the View to the different players in the application. The left mouse button corresponds to the Tracker, the right button attaches to the lead Plane, and the middle button moves the View to the following Plane.

```
procedure Update (Instance : in out Object) is

   Input_Flags : Easy_Sim.Modifier.Standard_Input.Flags_Pointer;

begin

   --
   Input_Flags := Easy_Sim.Modifier.Standard_Input.Flag_Copy
      (Easy_Sim.Modifier.Standard_Input.Object
         (Easy_Sim.View.Modifier (Instance.Main_View.all).all));

   if    Input_Flags.Key_Down ('S') then
         Input_Flags.Draw_Stats := not Input_Flags.Draw_Stats;
         Input_Flags.Key_Down ('S') :=  False;

   elsif Input_Flags.Key_Down (Ascii.Esc) then
         Input_Flags.Quit_Program := True;
         raise Quit_Program;

   elsif Input_Flags.Left_Mouse_Down then
         Input_Flags.Left_Mouse_Down := False;

         Easy_Sim.View.Manager.Set_Player
            (View_Manager (Instance).all,
             Of_View   => Instance.Main_View,
             To_Player => Instance.Tracker);

   elsif Input_Flags.Middle_Mouse_Down then
         Input_Flags.Middle_Mouse_Down := False;

         Easy_Sim.View.Manager.Set_Player
            (View_Manager (Instance).all,
             Of_View   => Instance.Main_View,
             To_Player => Instance.Wing_Man);

   elsif Input_Flags.Right_Mouse_Down then
         Input_Flags.Right_Mouse_Down := False;

         Easy_Sim.View.Manager.Set_Player
            (View_Manager (Instance).all,
             Of_View   => Instance.Main_View,
             To_Player => Instance.Flight_Lead);
   end if;

   if Input_Flags.Draw_Stats then
      Performer_Pf.PfDrawChanStats
         (Easy_Sim.View.Channel (Instance.Main_View.all));
   end if;

end Update;
```

Figure 34. Circling_Planes Class Update Procedure

User input results in the setting of global input flags within the Easy_Sim.Modifier.Standard_Input package. The Update procedure calls Flag_Copy to gain access to the input flags and monitors those in which it is interested. When one of these flags is set, Update generally resets it and acts appropriately. For the mouse operations, this action is to call the View_Manager to switch the Player to which the View is attached.

5.1.4 The Application_Driver Program

With all of the organization accomplished in the Circling_Planes package, the driver has little left to do. The *Application_Driver* procedure is shown in Figure 35. The driver declares the Simulation object, whose initialization is performed automatically by constructor, and then calls Render on Simulation. This operation does not complete until the user hits the Escape key.

```
with Circling_Planes;
procedure Application_Driver is
   Simulation : Circling_Planes.Object;
begin
   Circling_Planes.Render (Simulation);
end Application_Driver;
```

Figure 35. Circling_Planes Simulation Driver Procedure

The Circling Planes example has shown the development of one specific Easy_Sim application. The following section provides more general guidance for developing applications.

5.2 General Application Development

This section describes the entry points in the Easy_Sim software architecture where different types of functionality may be added to enhance a simulation. Many of these concepts are also presented in earlier chapters, but they are summarized here. The abstract classes are analyzed first because their customization is necessary to produce a working application. This section then analyzes the concrete basic classes, and it finishes by covering specializations of the manager classes.

The most basic abstract class is the Player class. It provides the basic component for differentiating entity behavior within a simulation. It also supplies the architectural connectors that serve as the focal points for defining the different aspects of that behavior. A Player's behavior can be simple, like the Plane examined last section, or it can be complex and consist of many different pieces. The Plane modeled in the Virtual Cockpit application, for instance, comprises radar capabilities, an inertial navigation system, a weapons delivery systems, a head's up display, and a throttle and stick, all in addition to correctly modeling aircraft flight dynamics [Sny93,68,Fig25].

This modular approach establishes a precedent, whereby a subclass can gather functionality from many smaller classes and organize this functionality in one place to suit the requirements of the architecture. This *building block* method makes creating components much more flexible and establishes a set of modules that are reusable both at the design and code levels. The building block approach is not limited to the Player class. It can be applied to any class within the Easy_Sim hierarchy.

The other basic abstract classes in the Easy_Sim architecture include the Environment and the Modifier. In the Circling Planes example, the Envi-

ronment class has already demonstrated the building block approach to making simulation backgrounds by combining the Horizon and Sun classes to form a Terrain subclass that adheres to the Environment's predefined structure. Similarly, application developers can extend the Modifier class to handle the user input into their applications. The possibility exists that these user input devices may be used to modify Player movement or control the Simulation as well, but these concepts are still under development. The abstract Draw operation in Environment forces its subclasses to declare any intentions of adding application specific features to the draw thread. Some Environments may be aided by grids or text overlays, and these features must be added during the draw thread.

The final abstract class in the Easy_Sim architecture, Simulation, is the hub of the Easy_Sim concept. Serving to coordinate the attributes that compose it, the Simulation supplies architectural connectors so that the application developer can isolate the location where organizational algorithms can be used in an application. As the Circling Planes example demonstrated, accomplishing the coordination during initialization simplifies the simulation processing later. Because the Simulation encompasses all other classes, it is revisited at the end of this section.

There are two basic concrete classes in the Easy_Sim architecture, of which Model is the more basic. The Model class forms a module isolating the representation of graphical images, and provides default behavior to process these images. Performer allows the Model class to be flexible enough to handle many different formats of images and automatically adjust for level of detail control, but a developer may still need to customize his Models. The most likely occurrence would be when the developer must work with a coordinate system

that differs from the Performer conventions. The current standards for distributed interactive simulations (DIS) fall into this category, and a Model subclass to correct for this discrepancy is warranted in DIS applications.

The other basic concrete class in the Easy_Sim architecture is the View class. A View encompasses window displaying, and it therefore also directs functions that are tied to windowing. These include user input and the ability to affect Performer's cull and draw threads, both of which an application developer may want to customize. User input can be customized through the Modifier class to accommodate different input devices, but the handling of the input data may fall more sensibly in the View class. Customized culling can enhance the application's performance, and the draw operation can add extra information to the scene, such as text overlays.

The manager classes in the Easy_Sim architecture are container classes, each of which directs the interaction among its constituents. The default Model_Manager prevents duplicate nodes in the simulation's rendering tree by cloning Models that are used more than once. This simple default design of the Model_Manager should suffice for most applications, but others may wish to customize how the Models in the simulation are manipulated. A Model_Manager subclass may wish to define different techniques for level of detail control, or for compatibility with a different coordinate system. A subclass may also simply require another avenue for assigning the Models.

The Player_Manager class also forms the architectural entry point for a limitless number of customizations for Easy_Sim applications. The default class provides no real organization of the Players, but different subclasses could institute methods for optimizing rendering by spatially organizing the Players [Har94,130]. An application could add collision detection into Easy_Sim appli-

cations through the Player_Manager. Most importantly for distributed interactive simulations, the Player_Manager class provides a sound starting point for bringing network player managers into the architecture. Ideally, an abstract class could be designed that is general enough for any network simulation. Subclasses could be derived for each particular application, customizing its specific needs according to a standard, understandable architectural plan.

Like its other manager classes, Easy_Sim's View_Manager class provides default functionality, fully expecting that applications will customize their View management by inheritance. Window management will probably be the most widely used reason for tailoring the View_Manager class in applications, as each View has its own window in the display. The View_Manager class will also administer interactions between input devices, because user input is commonly obtained from windows through the operating system.

Finally, the Simulation class serves as the focal point of an application, defining how its different pieces interact and providing executive control over the program. A Simulation subclass must first define the Environment, Players, and Views that it uses, as well as the associations that interconnect them. A subclass must provide initialization techniques for its attributes and ensure that each piece of the simulation is properly incorporated into the application's scheme. Finally, a Simulation subclass must supply the algorithms that define the interactions among its attributes during the simulation.

This chapter has outlined the development of one application and described the general techniques used to inherit from the Easy_Sim framework to create an application. The next chapter looks at the different versions of the Easy_Sim and ObjectSim frameworks, and draws results from their comparison.

VI Results and Comparisons

This chapter presents an analysis of the different versions of the ObjectSim and Easy_Sim application frameworks. The data contained here shows the success of this research effort by demonstrating that a visual simulation system software architecture can be implemented with an Ada 9X application framework so that it provides capabilities comparable to a similarly designed C++ framework.

The first section of this chapter covers background material and compares two performance measurements, frame rate and application thread time. The second section shows the different sizes of the executable programs. The final portion of the chapter examines differences in language features between the Ada 9X and C++ versions of the Easy_Sim framework.

6.1 Performance Comparisons

The most important performance measurement in any visual simulation system is its *frame rate*, or the number of individual screen images the simulation displays per second. This number indicates the realism portrayed to the system's users, as the illusion of motion tricks the human eye when individual frames are presented in rapid succession. If the frame rate becomes too slow, the viewer begins to notice individual frame boundaries, a phenomenon known as *jitter*.

The frame rate considered adequate for a simulation depends on the purpose of the simulation system, the expectations of its user, and the method of display. A person being entertained demands total realism, while trainees tolerate some jitter in order to accomplish their objectives. Additionally, the

consequences of jitter are multiplied if the viewers are totally immersed in display, as the stability offered by their peripheral vision is absent.

Taking these variables into account, jitter becomes noticeable near thirty frames per second. The ideal frame rate for simulations is sixty frames per second, the rate at which most televisions operate. The AFIT Graphics Lab considers twelve to fifteen frames per second acceptable for its simulations, as the users accept some inaccuracy in a research environment. This rate varies somewhat depending on the type of viewing device used.

This research effort used functions provided by SGI's Performer library to gather and display performance statistics. The data collected includes the frame rate and the time spent per frame in each of the application, cull, and draw threads (see Section 2.4). Because an application is only rendered as fast as its slowest process, the thread times are important in determining if one process impedes the entire application. Analysis of the application thread shows the differences among the different versions of the implementation, because the application developer's code is executed in this thread. By default, Performer executes the cull and draw threads on its own.

A simple application, like the Circling Planes described in Chapter V, is *draw limited*, meaning that the draw thread takes much longer to execute than either the application or cull thread. The complex models used to represent the Planes and Terrain, contrasted with the simple movement of the Planes, account for this difference. Most applications are *application limited*, however, as the reproduction of realistic Player behavior is often quite intricate. The Space Modeler [Van94], for instance, models the movement of planets and stars, and performing the corresponding calculations each frame is many

times more computationally expensive than lighting the small areas on the screen that represent these objects graphically.

Throughout the development of the Easy_Sim application framework, much of the complex functionality of ObjectSim was not transferred. These features all contribute to the ObjectSim framework, but they are not essential in the production of a visual simulation. Unfortunately, the two frameworks cannot be compared fairly with this differing functionality. Anticipating this divergence, a C++ version of the Easy_Sim architecture was maintained to parallel the Ada 9X version. A comparison between these two versions of the Easy_Sim framework is made later in this chapter.

The tests gathering performance statistics were undertaken on a four-processor SGI *Onyx/Reality Engine*^{2} machine running at 150 megahertz. The tests were conducted under controlled conditions, with one user logged into the machine, and two open winterm windows. The application window was opened to full screen size to maximize the rendering area and provide consistency for each test. The machine uses version 5.2 of the *IRIX* operating system. Both the Ada 9X and C++ versions of the application framework were compiled with maximum optimization.

6.1.1 Frame Rate Comparisons

The implementation of Easy_Sim was originally tested by reproducing the first two ObjectSim example applications [Sny93,AppA] with both Easy_Sim and ObjectSim. However, because these examples are draw limited, and the ObjectSim and Easy_Sim implementations render the same scenes, the frame rates match exactly. These tests were therefore inconclusive, and computationally expensive applications were devised to provide a better comparison.

These new applications were based on the Circling Planes example, and merely prolonged its application thread by increasing the number of aircraft. This expansion resulted in a simulation in which multitudes of planes fly routes whose paths resemble the overlapping rings of the Olympic flag. The number of planes in the formation was first expanded to 10, then to 100, 400, and 1,000.

Each of these simulations can be viewed from three locations: the second plane on one end of the formation, the last plane on the other end, or the tracker who observes from afar. When viewed from the tracker, an intimidating line of planes moves across the screen. Unfortunately, this angle does not provide any better comparison data, because drawing the entire formation is still more costly than moving it. If the view is moved to one of the planes, the desired effect is achieved. From these angles, the plane cycles through points where the attached view sees all the other planes aligned and points where it sees no other planes. The first case remains draw limited, because rendering all of the planes is still arduous. However, the second case is application limited, as the simulation still performs calculations to move the planes even when they are hidden from view.

Given these standard applications, tests were run using both the Ada 9X and C++ versions of the Easy_Sim implementation. ObjectSim was also tested as a baseline reference point, even though its differing C++ functionality prevented a truly fair comparison. Table 2 shows the test results. For each of these frameworks, data was gathered on the four different quantities of aircraft. The first row for each framework shows the application limited test, when all planes are moving but *none* are being shown. The second row for

Table 2. Frame Rates for Circling Planes Simulations

Architecture/ Language	Planes Seen	Plane Positions Being Updated			
		10	100	400	1000
Easy_Sim/ Ada 9X	none	30	30	30	10
	all	20	12	4.0	1.7
Easy_Sim/ C++	none	30	30	30	10
	all	20	12	3.8	1.6
ObjectSim/ C++	none	30	30	30	12
	all	20	15	3.8	1.6
		(Frames per second)			

each framework shows the draw limited case, when *all* planes are moving and being drawn.

These results are remarkable because they suggest that there is no significant difference in the performance of the three versions at given levels of stress. Most important is the direct comparison between the Ada 9X and C++ versions of Easy_Sim, which have mirror-like implementations (see Section 6.3). The results demonstrate that the use of the Ada 9X language itself does not hinder the rendering of a visual simulation. The gathering of evidence that the Ada 9X version of Easy_Sim performs at a level comparable to the same application built in C++ succeeds in satisfying one of the main goals of this thesis effort.

The next section examines a more specific aspect of the performance by focusing on the application thread times of the Circling Planes simulation.

6.1.2 Application Thread Time Comparisons

While Table 2 above shows that the bottom line performance of the Ada 9X Easy_Sim framework is similar to both the ObjectSim and Easy_Sim/C++ versions, the frame rate encompasses the execution times of all three of the Performer threads (see Section 2.4). It is only the application thread, however that illustrates the differences between the different versions of the Circling Planes application, because Performer executes the overwhelming majority of the cull and draw threads. Table 3 shows the application thread times per frame for the Circling Planes applications. Because the application thread time does not depend on the view, only one row is necessary for each version of the application framework.

Table 3 clearly shows the performance difference of the Ada 9X code, which runs 25% slower than the corresponding C++ code. This extra processing is expected, and can mainly be attributed to the immaturity of the GNAT compiler. Just like this thesis, the GNAT team's first objective is to produce working code, with optimization a secondary long term goal. In fact, no work at all has been done on Ada specific optimizations [Dew94b]. Given this statement, the fact that the code is only 25% behind is remarkable. Furthermore, given that the bottom line performance of the simulation is equivalent with

Table 3. Application Thread Times for Circling Planes Simulations

	Plane Positions being Updated			
Architecture/Language	10	100	400	1000
Easy_Sim/Ada 9X	1.7	8.5	31	75
Easy_Sim/C++	1.4	6.5	25	60
ObjectSim/C++	7.1	11.4	26	61
	(Milliseconds per Frame)			

the compiler in this state, the future looks very bright for Ada 9X in the simulation industry.

Another common factor in the slower running of Ada code is the run-time checking associated with exception handling. The Ada 9X application framework for Easy_Sim follows the Ada mindset of using exception handling to make the code more reliable and easier to test, and this programming style greatly eased the development of the Easy_Sim framework. Regardless, the Circling Planes test cases were recompiled with the run-time checks suppressed to see how it would affect the performance of the code. Surprisingly, the effects were minimal. Once again, however, the compiler developers blame this behavior on the immaturity of their product. Because the checks in the run-time system of version 1.83 have not been suppressed, suppressing the checks in a source program will not affect its performance. This hindrance has been removed from version 2.00 [Ban94].

The ObjectSim row of Table 3 is interesting for two reasons. Somewhere ObjectSim incurs processing overhead in its applications, most likely because of its use of shared memory. Using this Performer feature is costly for small applications, but the initial investment is returned when the simulation becomes complex. The test results also demonstrate that the efficiency supposedly gained by ObjectSim's rejecting encapsulation and relying on global access is not perceptible, as both C++ frameworks slow at an equal rate when the processing load is increased. This result is welcome news for supporters of the software engineering discipline. Programmers in performance critical software fields have often dismissed the principles of software engineering by citing efficiency concerns, but these results indicate that similar efficiency can be attained when following a well-planned, encapsulated design. Just as

the immaturity of the GNAT compiler can be blamed for the slowness of the Ada code, the maturity of the optimization techniques of the AT&T C++ compiler can be thanked for unraveling encapsulation on the machine code end.

The performance issues analyzed in this section strongly suggest that Ada 9X can perform alongside the C-based languages. Ada's shortcomings in the application thread time are well worth the development headaches it avoids. This notion is especially true considering that the bottom line performance is not affected and future gains in compiler optimizations are assured. The next section also analyzes an area in which Ada is notorious--code size.

6.2 Executable Size Comparisons

With the constantly expanding space available on today's computer systems, the importance of program size is waning rapidly. However, this section quickly examines the issue for a complete analysis.

Ada programs have traditionally been larger than C and C++ programs. The reasons for this dilemma included difficulty in optimizing the complexity of the language features, the run-time checking mentioned above, and the inclusion of large libraries for such tasks as input and output. Recently, Ada 83 compiler maturity has alleviated this problem [Law92].

Table 4 shows the sizes of the executable files for the Circling Planes example programs. Because the different versions of Circling Planes only vary in the length of an array, the 10, 100, 400, and 1,000 plane versions are all virtually the same size. Generally, the Ada 9X versions are just under twice as large as the C++ versions.

Table 4. Sizes of Executable Code for Circling Planes Simulations

Architecture	Language	Features	Kbytes
Easy_Sim	Ada 9X	Optimized, Inlined	1186
Easy_Sim	Ada 9X	Optimized	1185
Easy_Sim	Ada 9X	Totally Regular	1189
Easy_Sim	Ada 9X	Inlined	1211
Easy_Sim	Ada 9X	Optimized, Inlined, Suppressed	1164
Easy_Sim	Ada 9X	Optimized, Inlined, No Text_IO	1174
Easy_Sim	C++	Optimized, Some Inlining	703
ObjectSim	C++	Optimized, Some Inlining	666

The table shows six different versions of the Ada 9X examples, showing experimentation with different compilation options. The version whose performance statistics were presented in the last section is the inlined and optimized version, shown at the top. The inlining was applied to the Get and Set operations, as described in Section 4.3. Varied compilation options were analyzed originally because the AT&T C++ compiler would not support inlining to the extent of the GNAT compiler. To ensure that the Ada version's more prominent use of inlining did not sway the results, the application framework was rewritten without using the inline pragma, and all four tests were run again. No appreciable differences were found in the performance of the non-inlined version, and the size of the code remained virtually the same.

This result is slightly surprising, but it is more understandable when the maturity of the compiler is again considered. The machine code optimization of the established GNAT gcc back end is mature enough to automatically inline the simplistic calls within the same module. However, the GNAT 1.83

front end has not yet tackled inlining across package boundaries [Dew94a]. Therefore, the optimization originally intended by the use of the pragma was never being completely realized. The pragmas are therefore redundant when the compiler's optimization techniques are employed, as the data in the first two rows of Table 4 indicates.

A more normal increase in the size of the inlined version is evident in the fourth row of the table, when optimization is not used. As expected, the performances of the versions run without optimization are a notch slower than their optimized counterparts. The application thread time of the non-optimized inlined Ada 9X version runs 88 milliseconds for 1,000 circling planes, as compared with 71 milliseconds when optimized. This extra time translates into a frame rate of 8.6, instead of the optimized version's 10 frames per second.

Additional data was collected on two other versions of the Easy_Sim framework running the Circling Planes tests. The first was compiled with all run-time checks suppressed. This version lessened the code by 22 kilobytes, or 2%. Removing the use of the Text_IO and thereby all of the information displayed on the console removed 12 kilobytes from the code size.

Because of the lessening importance on code size, this problem has been a very low priority for the GNAT developers. They do plan, however, to use dynamic linking sometime in the future. Because this technique "can make a significant difference," the potential does exist for GNAT to become more competitive with the C languages in the arena of code size [Com94].

The next section discusses the similarities and differences between the Ada 9X and C++ implementations of the Easy_Sim application framework.

6.3 Language Comparisons

As shown throughout this chapter, a C++ version of the Easy_Sim application framework was maintained with the same functionality as the Ada 9X version, so that objective performance comparisons could be made between the two languages. This section compares and contrasts the more subjective features of the languages themselves.

The C++ version of the Easy_Sim framework was constructed to match the Ada 9X version as closely as possible, again to provide a fair performance comparison. While this approach seems like it would neglect important features of C++ design, the two languages have actually converged so closely that the major object-oriented features of C++ are employed heavily. In other words, a C++ implementation of the Easy_Sim software architecture developed straight from the design would look almost identical to the version produced by translating the Ada 9X implementation.

Each class package in Ada 9X became a class in C++, with constructors, destructors, a Configure function, and various virtual functions. The private parts of the Ada code were generally placed in the class header's protected section, so that they could be accessed by client programmers deriving subclasses. Ada 9X's child packages were brought to the top level in C++. The rest was generally translated construct for construct, in a very straightforward manner, so that the frameworks ended extremely similar. The rest of this section points out the minor differences that arose.

Just like GNAT, the AT&T C++ compiler would not allow all supposedly legal constructs within the language to be implemented. As alluded to in the previous section, the compiler would not allow inlining within the source files. The C++ compiler also did not support the language's new exception

handling mechanisms, and the locations in the Ada 9X code where exceptions are used were implemented with conditional statements. Finally, the C++ compiler had problems discerning different uses of the same identifier. In the Ada version, a Get function returning a class variable generally bears the name of that class. The C++ compiler was confused by this ambiguity, however, and *Get* was therefore included explicitly in the names of all the Get functions.

While the unimplemented features of C++ were impossible to use, those features that were implemented are often more complicated than their Ada counterparts. Parameter passing is one of these cases. Because C++ stores arrays as constant pointers, a function cannot return an array value. This limitation caused all of the Get operations for the Positions and Directions to be implemented as void functions where the array in which the value was to be placed is passed by reference as an argument. This inconsistency in handling types causes the uniformity of the C++ version to suffer. Both Ada and C++ allow parameters to have default values, but this feature is easier to use in Ada. Because the use of named notation in Ada parameter passing allows specification of actual parameters in any order, not every parameter needs to be passed. C++ only allows the last parameters in a function's argument list to be skipped. This rule does not allow a client programmer to use the default value for the second last argument and provide a value for the last argument. Many of Easy_Sim's Configure operations are designed so that only one or two parameters of six or seven need be specified, but this design will not work in C++ given certain combinations of arguments. The C++ programmer will therefore have to provide values for variables she would rather ignore.

The Configure operation is one aspect of the Easy_Sim architectural design that might be implemented differently in C++. Because constructors in

C++ can have arguments, some of the Configure operations could be performed as overloaded constructors. However, many of the parameters to Configure are not known when the object is created, so some of the Configure operations would still have to be explicitly called after construction. In order to maintain consistency throughout the architecture, therefore, the solution used in the Ada 9X version is also the better solution in C++.

Controlled operations, Ada's counterpart to default constructors and destructors, also contain the *Adjust* procedure, whose functionality is absent in the C++ model. Adjust is used so that a client programmer can explicitly control assignment between two variables (see Section 2.1.2). The Ada 9X Easy_Sim application framework uses the Adjust procedure to conveniently clone different models, as this operation is called automatically when a model is assigned to another. The C++ version requires the explicit call of the Clone operation.

Lest it appear that this section is too Ada biased, the final feature discussed is easier to use in C++. Ada 9X package specifications have kept to the original Ada design of a public and private part. If the attributes of a class are encapsulated privately, they are not automatically visible within derived subclasses. If a child class would like access to the private part of its parent, it must be declared inside a hierarchical library unit that is a child of the class package. This model is not always desirable, however. First, private parts are no longer that private. Second, the components of the Easy_Sim framework are all defined to be part of the Easy_Sim package, and application developers are encouraged to derive subclasses of the Easy_Sim components in their own packages. The subclasses therefore do not have access to their parent's attributes (see Section 5.1.3). Protected parts of C++ header files solve this

dilemma by specifying that subclasses are allowed visibility of the parent's member data. This model is much cleaner and preserves the privacy model necessary to ensure proper encapsulation.

This chapter has compared the different versions of the Easy_Sim application framework by analyzing both performance and language issues. The next chapter concludes this research effort and suggests areas of focus for future studies.

VII Conclusions and Future Study

The development of Easy_Sim has improved the ObjectSim software architecture for visual simulation systems and been instrumental in showing that the object-oriented features of Ada 9X can compare with their counterparts in C++. There is still much more potential for these languages to work together, as well as for the added benefits of Ada 9X to be realized. This chapter reviews the accomplishments of this thesis effort, makes suggestions for areas of future study, and outlines methods for the reader to obtain more information on Easy_Sim.

7.1 Easy_Sim and Its Accomplishments

Easy_Sim is both a software architecture and an application framework realizing that architecture. At the design level, Easy_Sim follows the object-oriented data abstraction model defined by Garlan and Shaw [Garl93,7-8] (see Section 2.2). Easy_Sim provides the structure by which developers can create their applications, allowing them to exploit a proven basic design and behavior for their systems.

Easy_Sim improves upon its predecessor, ObjectSim, in many ways. Where ObjectSim tried to present a coherent, encapsulated, and consistent design, it fell prey to productivity demands. The result was a framework that is difficult to understand, arduous to use, and troublesome to modify. Without the pressure of dependent projects, Easy_Sim's architecture was able to evolve into the design that was envisioned for ObjectSim. Dr. Jean Ichbiah, the original architect of Ada, has said, "I am driven by aesthetic considerations and the strong belief that only beautiful shape can be correct shape" [Ich92].

Easy_Sim follows this tenet, and creates an evenly formed architecture and implementation whose consistency throughout differing levels of abstraction breeds simplicity and ease of use.

This thesis effort has produced more than just aesthetically pleasing code, however. By backing up the architecture with data demonstrating the framework's production of efficient applications, Easy_Sim has shown that good designs can also be successfully implemented. Better yet, Easy_Sim has displayed this concept in the graphics field, a discipline which historically has shunned software engineering principles under the auspices of efficiency. Hopefully Easy_Sim has made inroads that will not be forgotten.

Easy_Sim has also been one of the seminal object-oriented projects to be implemented in Ada 9X. Many of the concepts originally thought to be trivial in the language proved more complex when scaled to a project the size of Easy_Sim. The lessons learned from Easy_Sim have already contributed to the object-oriented Ada community, and hopefully further Easy_Sim development will continue this trend.

7.2 Suggestions for Future Study

Although the Easy_Sim architecture and framework have succeeded in attaining their preliminary objectives, work remains to further their development. Many features implemented in ObjectSim were not carried into Easy_Sim, and no metrics have been collected to quantify the quality of either system. Both the architecture and framework of Easy_Sim could benefit from the development of a large scale application to test the varying uses of Easy_Sim classes proposed in Section 5.2. If successful, this project could serve as an example throughout the object-oriented, graphics, simulation, and Ada communities.

The following discussion of the areas for future work is broken into two sections. The first section outlines the architectural issues to investigate, and the second section lists areas in which improving the implementation would be beneficial.

7.2.1 Architectural Improvements

Many ideas that were entertained during the evolution of the Easy_Sim design were never realized, as the primary effort was focused on producing and evaluating a working system. Now that this endeavor has been accomplished, there is room to go back to the simulator industry and conspire to produce an architecture that defines the industry standard and is independent of the SGI platform. Individuals throughout the commercial sector have already expressed interest in helping to attack this problem.

As many techniques for general Easy_Sim application development have been proposed in this thesis (see Section 5.2), components need to be built to examine the feasibility of these techniques. For instance, is the Player_Manager class the right place to incorporate collision detection or network management capabilities? The construction of an Easy_Sim application can validate this question. The remainder of this section proposes specific areas of the Easy_Sim architecture where improvements can be made or alternative designs studied.

There are two areas in which the View_Manager needs to be analyzed further. Because Views provide a method of looking into a scene, they are often associated with display windows. Realizing this, an attempt was made to move the Open_Window operation from the Simulation class to the View_Manager, but an inexplicable compiler bug prevented the completion of

this architectural change. This preferred structure should be reassessed as the compiler matures. The View_Manager class also supports multiple View management, but this feature has not been tested by an application. Performing these tests can validate the model upon which the multiple View support is based.

Another manager class, the Player_Manager, also has yet to be tested to the extent of its design. In Sections 3.7 and 5.2, it was suggested that the Player_Manager serve as the basis for collision detection and distributed interactive simulation (DIS) communication for an application. Like the multiple Views, these design concepts also still need to be validated by demonstration.

Although it was not mentioned in Section 3.3, the ObjectSim Flt_Model class provides mechanisms to convert a Flt_Model established for the DIS coordinate systems so that they are compatible with Performer's coordinate system. While this functionality is necessary and useful for DIS applications, Easy_Sim endeavors to place it inside a subclass of the Model class instead of the Model itself. This approach allows an application developer to avoid the extra overhead if they do not require DIS capabilities, but provides a useful class if they do want to use it. Once again, a demonstration application can be developed to show off the capabilities of the framework.

The model used for handling user input within the Easy_Sim framework is based on ObjectSim and performed in the Modifier class. However, there is no reason that the input need be associated only with the View class, as the Modifier is intended. Players also need a standardized component with connectors to provide a common method for steering entities within a simulation. With this idea in mind, a standardized model for input should be developed,

taking into account that input can affect control of a View, control of a Player, or control of the Simulation itself. The most likely solution would be to provide a new Input class, through which the Modifiers could determine the user's intentions and pass the consequence of the input to Views or Players. The Simulation class could access the Input class directly to control the application. Suggestions for the implementation of this idea are developed further in the next section.

There are a couple of other design areas that may be implementation issues, but they are proposed briefly here. First, the Configure operation, which is used by Easy_Sim to provide parameterized initialization, was originally employed because controlled operations in Ada 9X do not allow parameters other than the controlled type itself. Some recent ideas have been put forth that propose to trick the compiler into allowing parameterization, however [Kem94]. These proposals should be evaluated to see if they might enhance both the Easy_Sim design and implementation. Second, Easy_Sim currently has no provisions for multiple Performer Pipes, each of which represents a multiprocessing system for the Performer threads. Providing the capability within the Easy_Sim architecture to support this capability opens an avenue for limitless expansion of computing power, as the hardware for visual simulations to exploit continually advances.

Having provided numerous areas in which improvements in the Easy_Sim software architecture might occur, discussion now focuses on aspects of the Easy_Sim implementation that might be improved. Both Performer and Ada 9X issues are presented.

7.2.2 Implementation Improvements

Just as the design of Easy_Sim could benefit from more in-depth analyses, so could the implementation. Because not all of the ObjectSim functionality was carried into Easy_Sim, migrating the rest of this functionality would be a good starting place for enhancing the Easy_Sim framework. These features include traversal masks, jitter removal, shared memory use, and round earth coordinate utilities. The rest of this section suggests improvements to the Performer and Ada 9X aspects of the code.

There are several areas where the use of Performer could be enhanced in the Easy_Sim framework. The most critical and probably most beneficial of these is the collection of user input. Currently this task is performed by GL functions. However, these capabilities will be removed in the next generation of GL, and the current Performer documentation suggests that all user input be handled by X Windows utilities [Har94,321].

Another area for improved use of Performer is the use of a texture list to preload all of the complicated textures into the application's memory. If this step is not performed, each time the simulation encounters a texture for the first time, the application halts while the database information for that texture is loaded. This hiccuping should be avoided if possible.

There are areas where communication between Performer and Ada 9X must occur. The callback operations for the cull and draw threads, as well as the window opening callbacks, currently do not have the ability to access any Ada variables, either through parameters or globally. For the cull and draw operations, the use of Performer's Passthrough Data capabilities may solve the problems [Har94,140-143]. Alternate solutions may be necessary for the win-

dow opening operation, however. If this solution works in all cases, then a consistent approach should be used throughout the entire framework.

There are some general coding concerns that Ada can address in Easy_Sim. The first and perhaps most visible of these is the repetition of the coordinate functions in Player, View, and Modifier. This massive code duplication can be avoided through the use of a generic package, but may also be corrected with tagged types. Either way the coordinates should probably be put in their own package and separated from the parent Easy_Sim package in which they are currently declared. The use of coordinates might also be enhanced by the use of subtyping for the heading, pitch, and roll. This step was not carried out however, because it is unclear that all applications would want to interpret these ranges similarly. For instance, should a pitch of 91 wrap around to 89 with the heading reversed? Or is it all right to have a roll of 450? This change should be carefully examined and justified if it is undertaken.

The Manager classes present a few interesting possibilities for code improvement. Because they have similar foundations, it may be possible to make a base generic package from which they could all be instantiated. The Manager classes might also be able to profit from the use of Ada 9X's protected types, which are a form of monitor [Bar93, KM94]. Because each class maintains a single list of objects, and access to those objects should be protected against multiple access problems in SGI's multiprocessor environments, the Managers seem a perfect fit for this new language construct. The management of the View states in the View Manager is rather complicated, and may stand to be split from the rest of the class. Because the application developer would have no reason to access this class package, it could be implemented as a

private package visible only within the Easy_Sim application framework hierarchy.

Possible design changes for the handling of user input were proposed above. Better solutions for the implementation of this area of the program are also needed. In the current design user input is recorded in flags contained in Modifier subclasses, and these flags are accessed globally by the application, the View, or whoever else might need them. This global use of the input violates the design of the Modifier class, which is only intended to be used by the View class. Hopefully a design solution will find a way to handle input that is not reliant on global access or the Modifier, possibly with the addition of an Input class as discussed in the previous section. This approach would allow the flags to be kept in a package by themselves, and this package should clearly denote that it is meant to be used as a global repository. Naming it Easy_Sim.Global_Flags or a similar name would at least make its purpose dreadfully understandable. The flags package would also be a textbook example of a case where a protected type could be used to guarantee that the many different classes in an application that access the global data do so safely.

Along the vein of input discussions, the Standard_Input class as currently implemented is monolithic, as it contains functionality for the mouse, keyboard, and keypad which could all be separated into building blocks. This disassociation would allow better modularity and flexibility, as an application developer could choose to use any combination of the standard devices.

The dependencies on Performer may be completely obscured in the Ada package specifications by liberal use of subtyping. This technique has been demonstrated by the Coords type, and could be instrumental in making the interfaces for the Easy_Sim framework platform independent.

Finally, the use of Finalization to perform memory reclamation when dynamically allocated objects are no longer needed has not been thoroughly examined. Although this neglect was in part due to Finalization's immature implementation by GNAT, this concept should be used in Easy_Sim so that memory management on the heap is as efficient as possible.

This section has described many areas in which the implementation of the Easy_Sim application framework might be improved by future researchers. The next section tells the reader where more information on Easy_Sim can be found, and it concludes this thesis.

7.3 Conclusion

It has been a pleasure to describe the work done in developing Easy_Sim. The task was certainly challenging and rewarding, and the interest of the reader is sincerely appreciated. The code for the Easy_Sim framework and a version of this document are available via anonymous ftp from archive.afit.af.mil in the pub/jkayloe directory. This directory can also be accessed through the World Wide Web at ftp://archive.afit.af.mil. If more information is needed or there are any questions, please contact the author directly.

Bibliography

[Abo93] Gregory D. Abowd, Len Bass, Larry Howard, and Linda Northrop, "Structural Modeling: An Application Framework and Development Process for Flight Simulators," Software Engineering Institute Technical Report CMU/SEI-93-TR-14, Pittsburgh, PA: Carnegie Mellon University, August 1993.

[And93] Christine Anderson and Erhard Ploedereder, "Managing the Transition to Ada 9X," *TRI-Ada '93 Conference Tutorials Volume 1*, Seattle, WA: Association for Computing Machinery, Inc., September 1993.

[ASC93] United States Air Force, "An Introduction to Structural Models," Technical Report USAF ASC-TR-93-5008, Wright-Patterson AFB, OH: Aeronautical Systems Center, 1993.

[Atk93] Colin Atkinson and David G. Weller, "Integrating Inheritance and Synchronization in Ada 9X," *TRI-Ada '93 Conference Proceedings*, Seattle, WA: Association for Computing Machinery, Inc., September 1993.

[Bal93] Brad Balfour, "Object-Oriented Programming Using Ada 9X," *TRI-Ada '93 Conference Tutorials Volume 3*, Seattle, WA: Association for Computing Machinery, Inc., September 1993.

[Ban92] Bernard Banner and Edmond Schonberg, "Assessing Ada 9X OOP: Building a Reusable Components Library," *TRI-Ada '92 Conference Proceedings*, Orlando, FL: Association for Computing Machinery, Inc., November 1992.

[Ban94] Bernard Banner, "Support for Pragma Suppress," Electronic Mail Message, New York: GNU New York University Ada 9X Translator Project, 2 December 1994.

[Bar93] John G. P. Barnes, *Introducing Ada 9X*, Office of the Under Secretary of Defense for Acquisition, 3 February 1993.

[Bar94] Stephane Barbey, Magnus Kempe, and Alfred Strohmeier, "Advanced Object-Oriented Programming with Ada 9X," *TRI-Ada '94 Conference Tutorials (Sunday)*, Baltimore, MD: Association for Computing Machinery, Inc., November 1994.

[Bei94] John Beidler, "Building Data Structure Components: A Case Study in Making the Transition from Ada 83 to Ada 9X," *Proceedings of the Eighth Annual ASEET Symposium*, Albuquerque, NM: January 1994.

[Boo94] Grady Booch and Doug Bryan, *Software Engineering with Ada*, Redwood City, CA: Benjamin/Cummings, 1994.

[Cer93] Gary J. Cernosek, "ROMAN-9X: A Technique for Representing Object Models in Ada 9X Notation," *TRI-Ada '93 Conference Proceedings*, Seattle, WA: Association for Computing Machinery, Inc., September 1993.

[Coh93] Norman H. Cohen, "Ada 9X as a Second Ada," *TRI-Ada '93 Conference Tutorials Volume 3*, Seattle, WA: Association for Computing Machinery, Inc., September 1993.

[Com94] Cyrille Comar, "Support for Pragma Suppress," Electronic Mail Message, New York: GNU New York University Ada 9X Translator Project, 2 December 1994.

[Coo92] Laura Cooper, *IRIS Performer Man Pages*, Mountain View, CA: Silicon Graphics, Inc., 1992.

[Cri92] Robert G. Crispen, Brett W. Freemon, K. C. King, and William V. Tucker, "DARTS: A Domain Architecture for Reuse in Training Systems," *Fourteenth Inter-Service/Industry Training Systems and Education Conference Proceedings*, November 1992.

[Cri94] Robert G. Crispen and Lynn D. Stuckey, Jr., "Structural Model: Architecture for Software Designers," *TRI-Ada '94 Conference Proceedings*, Baltimore, MD: November 1994.

[Dew94a] Robert Dewar, "Pragma Inline and Overloaded Subprograms," Electronic Mail Message, New York: GNU New York University Ada 9X Translator Project, 10 October 1994.

[Dew94b] Robert Dewar, "Support for Pragma Suppress," Electronic Mail Message, New York: GNU New York University Ada 9X Translator Project, 2 December 1994.

[Dia94] Milton E. Diaz, *The Photo Realistic AFIT Virtual Cockpit*, MS Thesis AFIT/GCS/ENG/94D-02, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH: December 1994.

[Dis92] Gary Dismukes, "Position Paper on Ada 9X and OOP," *TRI-Ada '92 Conference Proceedings*, Orlando, FL: Association for Computing Machinery, Inc., November 1992.

[Emb94] Wesley Embry and John Templeton, *The Ada 9X Paintball Demo*, Mountain View, CA: Silicon Graphics, Inc., September 1994.

[Epp94] Robert Epps, Telephone Interview, Binghamton, NY: CAE-Link Corporation, February 1994.

[Eri93] Matthew N. Erichsen, *Weapon System Sensor Integration for a DIS v2.0.3 Compatible Virtual Cockpit*, MS Thesis AFIT/GCS/ENG/93D-07, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH: December 1993.

[Feu82] Allan R. Feuer, *The C Puzzle Book*, Englewood Cliffs, NJ: Prentice Hall, 1982.

[For94] Jonathan L. Fortner, *Distributed Interactive Simulation Virtual Cassette Recorder: A Datalogger with Variable Speed Replay*, MS Thesis AFIT/GE/ENG/94D-10, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH: December 1994.

[Gard93] Michael T. Gardner, *A Distributed Interactive Simulation Based Remote Debriefing Tool for Red Flag Missions*, MS Thesis AFIT/GCS/ENG/93D-09, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH: December 1993.

[Garl93] David Garlan and Mary Shaw, *An Introduction to Software Architecture*, Advances in Software Engineering and Knowledge Engineering, Volume I, World Scientific Publishing Company, 1993.

[Ger93] William E. Gerhard, Jr., *Weapon System Integration for the AFIT Virtual Cockpit*, MS Thesis AFIT/GCS/ENG/93D-10, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH: December 1993.

[Gro92] David C. Gross and Lynn D. Stuckey, Jr., "Is Object-Oriented Design Sound Simulator Software Engineering?" *Fourteenth Inter-Service/Industry Training Systems and Education Conference Proceedings*, November 1992.

[Har94] Jed Hartman and Patricia (McLendon) Creek, *IRIS Performer Programmer's Guide*, Mountain View, CA: Silicon Graphics, Inc., 1994.

[Ich92] Jean D. Ichbiah, "A Farewell to Ada With Null," Electronic Mail Message to Christine Anderson, 20 November 1992.

[Kam93] J. Michael Kamrad, "Ada 9X: The Next Generation," *TRI-Ada '93 Conference Tutorials (Addendum)*, Seattle, WA: Association for Computing Machinery, Inc., September 1993.

[Kay94] Jordan R. Kayloe and Patricia K. Lawlis, "Easy_Sim: Using Ada 9X in a Graphics System Software Architecture", *TRI-Ada '94 Conference Proceedings*, Baltimore, MD: Association for Computing Machinery, Inc., November 1994.

[Kem94] Magnus Kempe, "Abstract Data Types are Under Full Control with Ada 9X," *TRI-Ada '94 Conference Proceedings*, Baltimore, MD: Association for Computing Machinery, Inc., November 1994.

[Ker78] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

[Kes94] Jim E. Kestermann, *Immersing the User in a Virtual Environment: The AFIT Information Pod Design and Implementation*, MS Thesis

AFIT/GCS/ENG/94D-13, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH: December 1994.

[Kun93] Andrea A. Kunz, *A Virtual Environment for Satellite Modeling and Orbital Analysis in a Distributed Interactive Simulation*, MS Thesis AFIT/GCS/ENG/93D-14, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH: December 1993.

[Law92] Patricia K. Lawlis and Terence W. Elam, "Ada Outperforms Assembly: A Case Study," *TRI-Ada '92 Conference Proceedings*, Orlando, FL: Association for Computing Machinery, Inc., November 1992.

[Law94] Patricia K. Lawlis and Mark I. Snyder, "An Object-Oriented Software Architecture for Large Scale Reuse," *Software Technology Conference Proceedings*, Salt Lake City, UT: Software Technology Center, 1994.

[McL91] Patricia McLendon, *Graphics Library Programming Guide*, Mountain View, CA: Silicon Graphics, Inc., 1991.

[McL92] Patricia McLendon, *IRIS Performer Programmer's Guide*, Mountain View, CA: Silicon Graphics, Inc., 1992.

[Mul92] *MultiGen Modeler's Guide*, San Jose, CA: Software Systems, December 1992.

[Pla89] P. J. Plauger, "Language Derby", *Embedded Systems Programming*, November 1989.

[Pla94] P. J. Plauger, "Ada for Large Systems?" Electronic Mail Message, 14 November 1994.

[Poh93] Ira Pohl, *Object-Oriented Programming Using C++*, Redwood City, CA: Benjamin/Cummings, Inc., 1993.

[Qui94a] Thomas J. Quiggle, "Calling C+- Member Functions," Electronic Mail Message, Mountain View, CA: Silicon Graphics, Inc., 12 September 1994.

[Qui94b] Thomas J. Quiggle, "SGI Inheriting C++ Classes," Usenet Posting to *comp.lang.ada* News Group, Mountain View, CA: Silicon Graphics, Inc., 18 November 1994.

[Rat92] Ada 9X Mapping/Revision Team, *Ada 9X Mapping Rationale*, Cambridge, MA: Intermetrics, Inc., March 1992.

[Rat93] Ada 9X Mapping/Revision Team, *Ada 9X Rationale*, Draft Version 4.0, Cambridge, MA: Intermetrics, Inc., September 1993.

[Rat94] Ada 9X Mapping/Revision Team, *Ada 9X Rationale*, Draft Version 5.0, Cambridge, MA: Intermetrics, Inc., 8 June 1994.

[RM93] Ada 9X Mapping/Revision Team, *Ada 9X Reference Manual*, Draft Version 4.0, Cambridge, MA: Intermetrics, Inc., 15 September 1993.

[RM94] Ada 9X Mapping/Revision Team, *Ada 9X Reference Manual*, Draft Version 5.0, Cambridge, MA: Intermetrics, Inc., 1 June 1994.

[Roh94] J. J. Rohrer, *Design and Implementation of Tools to Increase User Control and Knowledge Elicitation in a Virtual Battlespace*, MS Thesis AFIT/GCS/ENG/94D-20, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH: December 1994.

[Ros75] D. T. Ross, J. B. Goodenough, and C. A. Irvine, "Software Engineering: Process, Principles, and Goals," *Computer*, May 1975.

[Rum91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*, New York: Prentice-Hall, Inc., 1991.

[Rus94] James E Russell, *Multiple Model Adaptive Estimation and Head Motion Tracking in a Virtual Environment: An Engineering Approach*, MS Thesis AFIT/GCS/ENG/94D-21, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH: December 1994.

[Sch92] Edmond Schonberg, "Contrasts: Ada 9X and C++," Albuquerque, NM: Ada 9X Project Office, 22 April 1992.

[SEI93] Software Engineering Institute, "Structural Modeling Guidebook (Draft)," Pittsburgh, PA: Carnegie Mellon University, January 1993.

[She92] Steven M. Sheasby, *Management of Simnet and DIS Entities in Synthetic Environments*, MS Thesis AFIT/GCS/ENG/92D-16, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH: December 1992.

[She94] Steven M. Sheasby, Personal Interviews, Wright-Patterson AFB, OH: Distributed Simulation Technologies, Inc., October 1994.

[Sny93] Mark I. Snyder, *ObjectSim: A Reusable Object-Oriented DIS Visual Simulation*, MS Thesis AFIT/GCS/ENG/93D-20, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH: December 1993.

[Sny94] Mark I. Snyder, Electronic Mail Messages, Albuquerque, NM: Phillips Laboratory, Fall 1994.

[Sol93] Brian B. Soltz, *Graphical Tools for Situational Awareness Assistance for Large Synthetic Battle Fields*, MS Thesis AFIT/GCS/ENG/93D-21, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH: December 1993.

[Str86] Bjarne Stroustrup, *The C++ Programming Language*, Reading, MA: Addison-Wesley, 1986.

[Str91] Bjarne Stroustrup, *The C++ Programming Language: Second Edition*, Reading, MA: Addison-Wesley, 1991.

[Str94] Bjarne Stroustrup, "Ada for Large Systems?" Electronic Mail Message, Murray Hill, NJ: AT&T Research Laboratories, 13 November 1994.

[Taf91] S. Tucker Taft, "Multiple Inheritance in Ada 9X," Cambridge, MA: Intermetrics, Inc., 1991.

[Taf92a] S. Tucker Taft, "Ada 9X: A Technical Summary," *Communications of the ACM*, Vol. 35, No. 11, November 1992.

[Taf92b] S. Tucker Taft, "Panel on Ada 9X and Object-Oriented Programming," *TRI-Ada '92 Conference Proceedings*, Orlando, FL: November 1992

[Van94] John C. Vanderburgh, *Space Modeler: An Expanded, Distributed, Virtual Environment for Space Visualization*, MS Thesis AFIT/GCS/ENG/94D-23, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH: December 1994.

[Wil93] Kirk G. Wilson, *Synthetic Battle Bridge: Information Visualization and User Interface Design Applications in a Large Virtual Reality Environment*, MS Thesis AFIT/GCS/ENG/93D-26, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH: December 1993.

Vita

Captain Jordan Russell Kayloe was born on 16 June 1968 at Wright-Patterson Air Force Base, Dayton, Ohio. He spent his formative years in suburban Cleveland where he graduated from Strongsville High School in 1986. Kayloe attended Stanford University and graduated with a Bachelor of Science in Computer Science in June 1990. After his commissioning in the United States Air Force, Kayloe entered active duty in April 1991 and attended Basic Communications Officer Training at Keesler Air Force Base, Biloxi, Mississippi. He remained at Keesler afterwards with the 333rd Technical Training Squadron and served on its Mobile Training Team as an Ada and Software Engineering Instructor. During his tenure he trained over 150 students at 10 sites nationwide, until he was selected to attend the Air Force Insitute of Technology, where he enrolled in the School of Engineering in May 1993. Kayloe was promoted to Captain on 8 November 1994.

Permanent Address:
19482 Albion Rd.
Strongsville, OH 44136
kayloe@cs.stanford.edu

REPORT DOCUMENTATION PAGE

OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1994	Master's Thesis

4. TITLE AND SUBTITLE
EASY_SIM: A VISUAL SIMULATION SYSTEM SOFTWAREARCHITECTURE WITH AN ADA 9X APPLICATION FRAMEWORK

5. FUNDING NUMBERS

6. AUTHOR(S)

Jordan R. Kayloe, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
2950 P Street
WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/94D-11

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Mr. Donald J. Reifer
Chief, Ada Joint Program Office (AJPO)
701 South Courthouse Road
Arlington, VA 22204-2199

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Software architectures increase productivity when used as the basis for developing applications in a problem domain. This thesis describes the creation of Easy_Sim, an object-oriented software architecture for visual simulation systems, and its corresponding implementation as an application framework in Ada 9X. The research built upon ObjectSim, an existing object-oriented simulation architecture implemented as a C++ application framework. Both ObjectSim and Easy_Sim operate on Silicon Graphics platforms and use the IRIS Performer graphics programming library. Easy_Sim is implemented using version 1.83 of the GNAT compiler.

The investigation for this thesis involved honing ObjectSim's design, implementing the improved result in both C++ and Ada 9X, and developing applications to compare the two versions. The study achieved two main objectives: producing Easy_Sim as an improved visual simulation system architecture by building on ObjectSim's experience, and producing a visual simulation system application from Easy_Sim in Ada 9X that performs at a level comparable to the same application built in C++.

14. SUBJECT TERMS

Software Engineering, Ada 9X, Software Architecture, Application Framework, Object-Oriented,

15. NUMBER OF PAGES
160

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL